Programming with Inductive and Co-Inductive Types

John Greiner

January 27, 1992

CMU-CS-92-109

Carnegie Mellon

Programming with Inductive and Co-Inductive Types

John Greiner

January 27, 1992

CMU-CS-92-109

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We look at programming with inductive and co-inductive datatypes, which are inspired theoretically by initial algebras and final co-algebras, respectively. A predicative calculus which incorporates these datatypes as primitive constructs is presented. This calculus allows reduction sequences which are significantly more efficient for two dual classes of common programs than do previous calculi using similar primitives. Several techniques for programming in this calculus are illustrated with numerous examples. A short survey of related work is also included.

This research was supported in part by the Office of Naval Research and in part by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under Contract N00014-84-K-0415, ARPA Order No. 5404, and in part by the ONR Graduate Fellowship Program.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of ONR, DARPA or the U.S. government.

Keywords: Data types and structures, program and recursion schemes, lambda calculus and related systems, induction, co-induction, initial algebras, final co-algebras

1 Introduction

This paper explores programming with inductive and co-inductive datatypes. Type expressions using the μ type constructor represent finitely representable inductive types (e.g., natural numbers, lists), while those using ν represent countably infinite or potentially infinite co-inductive types (e.g., potentially infinite streams, infinite depth binary trees). Each (co-)inductive type is associated with operators to build and manipulate terms of these types.

Encodings of such types in previous calculi have suffered from efficiency problems which prevent them from being as useful in practice as desired. The typical example is that of the predecessor function on the common Church numeral encoding in F_2, which requires linear-time, as shown by Parigot [24]. Most encodings in other calculi are closely related and suffer the same problem. Our calculus allows similar encodings of data types, but the calculus offers extensions admitting definitions of constant-time *pred* and *cdr*, and related efficiency improvements.

Previous work has concentrated on inductive types. Very few examples of co-inductive functions have been given, so that their usefulness in practice is in question. This paper explores the duality of inductive and co-inductive types and presents a number of examples to attempt to show the usefulness of co-inductive types.

Induction and co-induction as presented here are obviously less powerful than recursion, as they guarantee *termination*. So, an essential question is whether these concepts are powerful enough for practical programming. This is the motivation for our extensive look at examples within the calculus.

These (co-)inductive type constructors are inspired by initial algebras and final co-algebras [8, 9, 19] of category theory. Some examples of this paper use simple commuting diagrams to help explain the definition of some functions in the calculus. A basic knowledge of category theory will be helpful, but not necessary, to the reader.

A summary of this theoretical motivation is given in Section 2. The calculus is introduced in section 3, with examples of inductive and co-inductive terms in Sections 4 and 5. In Section 6, related work is discussed. The appendices give additional details for interested readers.

2 Theoretical Inspiration

This section provides a theoretical motivation or inspiration for the calculus as presented in Section 3. As such, it also serves as a beginning point for developing a model for the calculus. More pragmatically, simple commuting diagrams lead to methods for programming in the calculus.

Assume that there exists an appropriate category C of types[1], where arrows

[1] Some features of such a category are discussed in [2, 6].

represent functions from one type to another and composition is function composition. We then look at F-algebras, for functors $F : \mathcal{C} \rightarrow \mathcal{C}$. An *$F$-algebra* is a pair $\langle \tau, g \rangle$ consisting of an object $\tau : \mathcal{C}$ and a map $g : F(\tau) \rightarrow \tau$. An *$F$-homomorphism*, an arrow in the category of F-algebras, is a $\mathcal{C}$-arrow such that the following diagram commutes.

$$\begin{tikzcd}
F[\tau] \arrow[r, "F(h)"] \arrow[d, "g"'] & F[\tau'] \arrow[d, "f"] \\
\tau \arrow[r, "h"'] & \tau'
\end{tikzcd}$$

If $\langle \tau, g \rangle$ is an *initial* F-algebra, then there exists a unique h for any given choice of $\langle \tau', f \rangle$. Labelling the initial F-algebra as $\langle \mu(F), \mathbf{in}\{F\} \rangle$ and the unique h as $\mathbf{R}\{F\}[\tau]f$ to emphasize their dependencies, we obtain the commuting diagram

$$\begin{tikzcd}
F[\mu(F)] \arrow[r, "F(\mathbf{R}\{F\}[\tau']f)"] \arrow[d, "\mathbf{in}\{F\}"'] & F[\tau'] \arrow[d, "f"] \\
\mu(F) \arrow[r, "\mathbf{R}\{F\}[\tau']f"'] & \tau'
\end{tikzcd}$$

When the above arguments are dualized, we obtain the *final F-co-algebra* $\langle \nu(F), \mathbf{out}\{F\} \rangle$ such that the following diagram commutes for any choice of $\langle \tau', f \rangle$.

$$\begin{tikzcd}
F[\tau'] \arrow[r, "F(\mathbf{G}\{F\}[\tau']f)"] & F[\nu(F)] \\
\tau' \arrow[u, "f"] \arrow[r, "\mathbf{G}\{F\}[\tau']f"'] & \nu(F) \arrow[u, "\mathbf{out}\{F\}"']
\end{tikzcd}$$

The following theorems and corollaries are to establish motivation for some of the equality judgments of the calculus. Proofs are given only for inductive cases, as the co-inductive cases are analogous.

The first theorem is β-like and shows the main interaction of the induction morphisms. The second gives η-like equalities.

Theorem 1 *Principle of induction:* $(\mathbf{R}\{F\}[\tau]f) \circ \mathbf{in}\{F\} = f \circ F(\mathbf{R}\{F\}[\tau]f)$ *and of co-induction:* $\mathbf{out}\{F\} \circ (\mathbf{G}\{F\}[\tau]f) = F(\mathbf{G}\{F\}[\tau]f) \circ f$.

Proof: Follow from the commutativity of the preceding diagrams. □

Theorem 2 $\mathbf{R}\{F\}[\mu(F)]\mathbf{in}\{F\} = Id^{\mu(F)}$ *and* $\mathbf{G}\{F\}[\nu(F)]\mathbf{out}\{F\} = Id^{\nu(F)}$.

Proof: Follow from the initiality (finality) of the F-(co-)algebra and commutativity. □

Theorem 3 $\mathbf{in}\{F\}$ *and* $\mathbf{out}\{F\}$ *are isomorphisms.*

Proof: We show that $\mathbf{in}\{F\}$ has left- and right-inverses. Consider the following diagram, where both squares commute:

$$\begin{tikzcd}
F[\mu(F)] \arrow[r, "F(!)"] \arrow[d, "\mathbf{in}\{F\}"'] & F[F[\mu(F)]] \arrow[r, "F(\mathbf{in}\{F\})"] \arrow[d, "F(\mathbf{in}\{F\})"] & F[\mu(F)] \arrow[d, "\mathbf{in}\{F\}"] \\
\mu(F) \arrow[r, "!"'] & F[\mu(F)] \arrow[r, "\mathbf{in}\{F\}"'] & \mu(F)
\end{tikzcd}$$

By the definition of an algebra, $! = \mathbf{R}\{F\}[F[\mu(F)]]F(\mathbf{in}\{F\})$ is the unique map such that

$$! \circ \mathbf{in}\{F\} = F(\mathbf{in}\{F\}) \circ F(!) = F(\mathbf{in}\{F\} \circ !).$$

Since the inner squares commute, so does the outer rectangle. Then, by initiality,

$$\mathbf{in}\{F\} \circ ! = Id^{\mu(F)}.$$

Furthermore,

$$! \circ \mathbf{in}\{F\} = F(\mathbf{in}\{F\} \circ !) = F(Id^{\mu(F)}) = Id^{F[\mu(F)]}.$$

But these equations simply mean that ! is a left- and right-inverse of $\mathbf{in}\{F\}$, and thus, that $\mathbf{in}\{F\}$ is an isomorphism. □

Corollary 1 *There exist unique morphisms* $\mathbf{in}^{-1}\{F\}$ *and* $\mathbf{out}^{-1}\{F\}$. *And these inverses are expressible in terms of the other (co-)inductive morphisms as* $\mathbf{R}\{F\}[F[\mu(F)]](F(\mathbf{in}\{F\}))$ *and* $\mathbf{G}\{F\}[F[\nu(F)]](F(\mathbf{out}\{F\}))$, *respectively.*

Example 1 For $F(X) = \mathbf{1} + X$, $\mathbf{in}\{F\}$ is the mapping $[\mathit{zero}, \mathit{succ}]$. Its inverse, $\mathbf{in}^{-1}\{F\}$, is related to the mappings *zero?* and *pred*. Details are found in Example 11.

3 The Calculus $\lambda^{MM\mu\nu}$

The calculus is based upon a restricted version of λ^{ML} [11]. The higher kinds of that calculus are omitted to avoid complications that arise between type constructor application and the positivity requirement of μ and ν[2]. The explicit *set* injection is also omitted, for readability. Thus, this calculus is called $\lambda^{MM\mu\nu}$, for "mini-λ^{ML} with μ and ν". The choice of calculus for a base is not critical; e.g., the Calculus of Constructions, F_2, and variants of F_2 have been used in other work.

3.1 Syntax

Given denumerable sets of variables *typevar* and *termvar*, the calculus is defined by

$$\begin{array}{rcl}
X \in \mathit{typevar} & & \\
\tau \in \mathit{types} & ::= & X \mid \mathbf{1} \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \tau \rightarrow \tau' \mid \mu(u) \mid \nu(u) \\
u \in \mathit{typecons}^{3} & ::= & \lambda X.\tau \qquad \text{s.t. } \neg \mathit{Neg}(X, \tau) \\
\sigma \in \mathit{typeschemes} & ::= & \tau \mid \sigma_1 \times \sigma_2 \mid \sigma_1 + \sigma_2 \mid \sigma \rightarrow \sigma' \mid \forall X.\sigma \\
x \in \mathit{termvar} & & \\
t \in \mathit{terms} & ::= & x \mid * \mid \langle t_1, t_2 \rangle \mid \pi_1 t \mid \pi_2 t \mid \\
& & \mathbf{inl}^{\sigma} t \mid \mathbf{inr}^{\sigma} t \mid \mathbf{case}(t, t_1, t_2) \mid \\
& & \lambda x : \sigma.t \mid t_1\ t_2 \mid \Lambda X.t \mid t[\tau] \mid \\
& & \mathbf{in}\{u\} \mid \mathbf{in}^{-1}\{u\} \mid \mathbf{R}\{u\}[\tau]t \mid \\
& & \mathbf{out}\{u\} \mid \mathbf{out}^{-1}\{u\} \mid \mathbf{G}\{u\}[\tau]t \\
\Gamma \in \mathit{contexts} & ::= & \emptyset \mid \Gamma, v : \sigma
\end{array}$$

where $\mathit{Neg}(X, \tau)$ ($\mathit{Pos}(X, \tau)$) holds if X occurs "negatively" ("positively") in τ:

[2] The implications are discussed briefly in Section 8.

[3] The weak notion of "type constructor" used here corresponds to a constructor of kind *Type* → *Type*. Also, μ and ν can be informally considered type constructors of kind (*Type* → *Type*) → *Type*.

$$
\begin{array}{l}
Neg(X,X) = Neg(X,\mathbf{1}) = \mathit{false} \\
Neg(X,\tau_1 \times \tau_2) = Neg(X,\tau_1 + \tau_2) = Neg(X,\tau_1) \vee Neg(X,\tau_2) \\
Neg(X,\tau \rightarrow \tau') = Pos(X,\tau) \vee Neg(X,\tau') \\
Neg(X,\mu(\lambda X.\tau)) = Neg(X,\nu(\lambda X.\tau)) = \mathit{false} \\
Neg(X,\mu(\lambda Y.\tau)) = Neg(X,\nu(\lambda Y.\tau)) = (X \neq Y) \wedge Neg(X,\tau),\ \text{if } X \neq Y
\end{array}
$$

$$
\begin{array}{l}
Pos(X,X) = \mathit{true} \\
Pos(X,\mathbf{1}) = \mathit{false} \\
Pos(X,\tau_1 \times \tau_2) = Pos(X,\tau_1 + \tau_2) = Pos(X,\tau_1) \vee Pos(X,\tau_2) \\
Pos(X,\tau \rightarrow \tau') = Neg(X,\tau) \vee Pos(X,\tau') \\
Pos(X,\mu(\lambda X.\tau)) = Pos(X,\nu(\lambda X.\tau)) = \mathit{false} \\
Pos(X,\mu(\lambda Y.\tau)) = Pos(X,\nu(\lambda Y.\tau)) = (X \neq Y) \wedge Pos(X,\tau),\ \text{if } X \neq Y
\end{array}
$$

and where contexts are taken to be sets, with the comma as the extension operator.

The addition of the families of constants $\mathbf{in}^{-1}\{u\}$ and $\mathbf{out}^{-1}\{u\}$, along with the association reductions in Section 3.4, is a primary contribution of this paper.

Here, we find the notation $\mu(\lambda X.\tau)$ (and similarly, $\nu(\lambda X.\tau)$) more convenient, but it is equivalent to the more common notation $\mu X.\tau$.

3.2 Meta-notation

For readability, we make extensive use of notational definitions using the symbol $\equiv$. For example, $Id^{\sigma} \equiv \lambda x : \sigma.x$, $t^{\dagger} \equiv \lambda u : \mathbf{1}.t$. Also, after the first examples, we omit type information when it is clear from context.

In the examples, we express desired properties of functions via equivalences. Observational equivalence between terms is denoted by $\cong$. As usual, $t \cong t'$ iff $P[t]$ and $P[t']$ evaluate to the same value[4], for all contexts $P[]$ of type *Bool*.

Capture-avoiding substitution is denoted $A[B/C]$. Type constructor application $u\ \tau$ is shorthand for its β-reduction, $u[\tau/X]$

3.3 Type judgments

Type judgments of the form $\Gamma \vdash t : \sigma$ state that t is a term of type σ.

$$
\Gamma, t : \sigma \vdash t : \sigma \qquad\qquad \Gamma \vdash * : \mathbf{1}
$$

$$
\frac{\Gamma \vdash t_i : \sigma_i \qquad i = 1,2}{\Gamma \vdash \langle t_1, t_2 \rangle : \sigma_1 \times \sigma_2}
\qquad\qquad
\frac{\Gamma \vdash t : \sigma_1 \times \sigma_2 \qquad \text{i=1,2}}{\Gamma \vdash \pi_i t : \sigma_i}
$$

$$
\frac{\Gamma \vdash t : \sigma_1}{\Gamma \vdash \mathbf{inl}^{\sigma_2} t : \sigma_1 + \sigma_2}
\qquad\qquad
\frac{\Gamma \vdash t : \sigma_2}{\Gamma \vdash \mathbf{inr}^{\sigma_1} t : \sigma_1 + \sigma_2}
$$

[4] See Section 3.4 for the definitions of evaluation and values.

$$\frac{\Gamma \vdash t : \sigma_1 + \sigma_2 \qquad \Gamma \vdash t_i : \sigma_i \rightarrow \sigma \qquad i = 1,2}{\Gamma \vdash \mathbf{case}(t, t_1, t_2) : \sigma}$$

$$\frac{\Gamma, x : \sigma \vdash t : \sigma'}{\Gamma \vdash (\lambda x : \sigma.t) : \sigma \rightarrow \sigma'} \qquad \frac{\Gamma \vdash t_1 : \sigma \rightarrow \sigma' \qquad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1\ t_2 : \sigma'}$$

$$\frac{\Gamma \vdash t : \sigma}{\Gamma \vdash \Lambda X.t : \forall X.\sigma} \qquad \frac{\Gamma \vdash t : \forall X.\sigma}{\Gamma \vdash t[\tau] : \sigma[\tau/X]}$$

$$\Gamma \vdash \mathbf{in}\{u\} : (u\ \mu(u)) \rightarrow \mu(u) \qquad \Gamma \vdash \mathbf{out}\{u\} : \nu(u) \rightarrow (u\ \nu(u))$$

$$\Gamma \vdash \mathbf{in}^{-1}\{u\} : \mu(u) \rightarrow (u\ \mu(u)) \qquad \Gamma \vdash \mathbf{out}^{-1}\{u\} : (u\ \nu(u)) \rightarrow \nu(u)$$

$$\frac{\Gamma \vdash t : (u\ \tau) \rightarrow \tau}{\Gamma \vdash \mathbf{R}\{u\}[\tau]t : \mu(u) \rightarrow \tau} \qquad \frac{\Gamma \vdash t : \tau \rightarrow u\ \tau}{\Gamma \vdash \mathbf{G}\{u\}[\tau]t : \tau \rightarrow \nu(u)}$$

Type and type constructor equality is simply α-equality.[5]

3.4 Operational semantics

The values of the calculus are the closed terms of the form

$$\begin{aligned} v \in \mathit{values} \quad ::= \quad & * \mid \langle v_1, v_2 \rangle \mid \mathbf{inl}^\sigma v \mid \mathbf{inr}^\sigma v \mid \lambda x : \sigma.t \mid \Lambda X.t \mid \\ & \mathbf{in}\{u\} \mid \mathbf{in}\{u\}v \mid \mathbf{in}^{-1}\{u\} \mid \mathbf{R}\{u\}[\tau]v \mid \\ & \mathbf{out}\{u\} \mid \mathbf{out}^{-1}\{u\} \mid \mathbf{out}^{-1}\{u\}v \mid \mathbf{G}\{u\}[\tau]v \mid \mathbf{G}\{u\}[\tau]v_1\ v_2 \end{aligned}$$

These are the values of the λ^{ML} core, plus the "partially applied" (co-)inductive forms.

The one-step evaluation function $\longrightarrow$ is defined upon well-typed terms by the following judgments. Its reflexive and transitive closure is denoted by $\overset{*}{\longrightarrow}$. Although lazy products and co-products would also improve efficiency, we use eager (co-)products and call-by-value which are sufficient for the desired efficiency results of induction and co-induction.

$$\frac{t_1 \longrightarrow t_1'}{\langle t_1, t_2 \rangle \longrightarrow \langle t_1', t_2 \rangle} \qquad \frac{t_2 \longrightarrow t_2'}{\langle v, t_2 \rangle \longrightarrow \langle t, t_2' \rangle} \qquad \frac{t \longrightarrow t' \qquad i = 1,2}{\pi_i t \longrightarrow \pi_i t'}$$

$$\frac{t \longrightarrow t'}{\mathbf{inl}^\sigma t \longrightarrow \mathbf{inl}^\sigma t'} \qquad \frac{t \longrightarrow t'}{\mathbf{inr}^\sigma t \longrightarrow \mathbf{inr}^\sigma t'}$$

[5] In particular, the calculus does *not* have the equality $\Gamma \vdash \mu(u) \approx u(\mu(u))$ as found in many object-oriented type systems using "recursive" types. Here, these two expressions denote distinct, but isomorphic types, where $\mathrm{in}\{u\}$ and $\mathrm{in}^{-1}\{u\}$ are coercions.

$$\frac{t \longrightarrow t'}{\mathbf{case}(t, t_1, t_2) \longrightarrow \mathbf{case}(t', t_1, t_2)}$$

$$\frac{t_1 \longrightarrow t_1'}{\mathbf{case}(v, t_1, t_2) \longrightarrow \mathbf{case}(v, t_1', t_2)}$$

$$\frac{t_2 \longrightarrow t_2'}{\mathbf{case}(v, v_1, t_2) \longrightarrow \mathbf{case}(v, v_1, t_2')}$$

$$\frac{t \longrightarrow t'}{t\ t'' \longrightarrow t'\ t''} \qquad \frac{t \longrightarrow t'}{v\ t \longrightarrow v\ t'} \qquad \frac{t \longrightarrow t'}{t[u] \longrightarrow t'[u]}$$

$$\frac{t \longrightarrow t'}{\mathbf{in}\{u\}t \longrightarrow \mathbf{in}\{u\}t'} \qquad \frac{t \longrightarrow t'}{\mathbf{in}^{-1}\{u\}t \longrightarrow \mathbf{in}^{-1}\{u\}t'}$$

$$\frac{t \longrightarrow t'}{\mathbf{out}\{u\}t \longrightarrow \mathbf{out}\{u\}t'} \qquad \frac{t \longrightarrow t'}{\mathbf{out}^{-1}\{u\}t \longrightarrow \mathbf{out}^{-1}\{u\}t'}$$

$$\frac{t \longrightarrow t'}{\mathbf{R}\{u\}[\tau]t \longrightarrow \mathbf{R}\{u\}[\tau]t'} \qquad \frac{t \longrightarrow t'}{\mathbf{G}\{u\}[\tau]t \longrightarrow \mathbf{G}\{u\}[\tau]t'}$$

$$\mathbf{case}(\mathbf{inl}^{\sigma} v, v_1, v_2) \longrightarrow v_1\ v \qquad \mathbf{case}(\mathbf{inr}^{\sigma} v, v_1, v_2) \longrightarrow v_2\ v$$

$$\pi_i\langle v_1, v_2\rangle \longrightarrow v_i$$

$$(\lambda x : \sigma.t)v \longrightarrow t[v/x] \qquad (\Lambda X.t)[u] \longrightarrow t[u/X]$$

$$\mathbf{in}^{-1}\{u\}(\mathbf{in}\{u\}v) \longrightarrow v \qquad \mathbf{out}\{u\}(\mathbf{out}^{-1}\{u\}v) \longrightarrow v$$

$$\mathbf{R}\{u\}[\tau]v_1\ (\mathbf{in}\{u\}v_2) \longrightarrow v_1(\Phi\{u\}[\mu(u)][\tau]\ (\mathbf{R}\{u\}[\tau]v_1)\ v_2)$$

$$\mathbf{out}\{u\}(\mathbf{G}\{u\}[\tau]v_1\ v_2) \longrightarrow \Phi\{u\}[\tau][\nu(u)]\ (\mathbf{G}\{u\}[\tau]v_1)\ (v_1\ v_2)$$

These last two rules represent infinite families of reductions indexed by u, since $\Phi\{u\}[\tau_1][\tau_2]\ f\ t$ is meta-notation defined by induction on the structure of u as below.[6]

Together, the type constructor u and $\Phi\{u\}[\tau_1][\tau_2]\ f\ t$ correspond to the object (type) morphism and map (term) morphism of the functor Φ of Section 2. Since the latter is defined in terms of the former it is sufficient to index the terms such as $\mathbf{in}\{u\}$ as such, rather than indexing by the functor as in category theory.

The definition of $\Phi\{u\}[\tau_1][\tau_2]$ is

[6] This definition is adapted in conjunction with Daniel Leivant and is a correction of the analogous definitions in Hagino [8, 9] and Leivant [12].

$$\begin{array}{ll}
\text{if } u \equiv \lambda X.X, & f\ t \\
\text{if } u \equiv \lambda X.Y, X \neq Y, & t \\
\text{if } u \equiv \lambda X.\mathbf{1}, & * \\
\text{if } u \equiv \lambda X.\tau_1' \times \tau_2', & \langle \Phi\{\lambda X.\tau_1'\}[\tau_1][\tau_2]\ f\ (\pi_1 t), \Phi\{\lambda X.\tau_2'\}[\tau_1][\tau_2]\ f\ (\pi_2 t)\rangle \\
\text{if } u \equiv \lambda X.\tau_1' + \tau_2', & \mathbf{case}(t,\ \lambda x_1 : \tau_1'[\tau_1/X].\mathbf{inl}(\Phi\{\lambda X.\tau_1'\}[\tau_1][\tau_2]\ f\ x_1), \\
 & \qquad\qquad \lambda x_2 : \tau_2'[\tau_1/X].\mathbf{inr}(\Phi\{\lambda X.\tau_2'\}[\tau_1][\tau_2]\ f\ x_2)) \\
\text{if } u \equiv \lambda X.\tau_1' \rightarrow \tau_2', & \lambda x : \tau_1'[\tau_2/X]. \\
 & \qquad \Phi\{\lambda X.\tau_2'\}[\tau_1][\tau_2]\ f\ (t(\overline{\Phi}\{\lambda X.\tau_1'\}[\tau_1][\tau_2]\ f\ x)) \\
\text{if } u \equiv \lambda X.\mu(u') & \mathbf{R}\{u'[\tau_1/X]\}[\mu(u'[\tau_2/X])] \\
\text{and } Y \text{ fresh,} & \quad (\lambda x : u'[\tau_1/X]\ \mu(u'[\tau_2/X]). \\
 & \qquad\quad \mathbf{in}\{u'[\tau_2\,'X]\} \\
 & \qquad\qquad (\Phi\{\lambda X.u'\ Y\}[\tau_1][\tau_2]\ f\ x)[\mu(u'[\tau_2/X])/Y]) \\
 & \quad t \\
\text{if } u \equiv \lambda X.\nu(u') & \mathbf{G}\{u'[\tau_2/X]\}[\nu(u'[\tau_1/X])] \\
\text{and } Y \text{ fresh,} & \quad (\lambda x : \nu(u'[\tau_1/X]). \\
 & \qquad\quad (\Phi\{\lambda X.u'\ Y\}[\tau_1][\tau_2] \\
 & \qquad\qquad f\ (\mathbf{out}\{u'[\tau_1/X]\}x))[\nu(u'[\tau_1/X])/Y]) \\
 & \quad t
\end{array}$$

where the definition of $\overline{\Phi}\{u\}[\tau_1][\tau_2]\ f\ t$ *is almost identical*, except that Φ and $\overline{\Phi}$ are interchanged:

$$\begin{array}{ll}
\text{if } u \equiv \lambda X.Y, X \neq Y, & t \\
\text{if } u \equiv \lambda X.\mathbf{1}, & * \\
\text{if } u \equiv \lambda X.\tau_1' \times \tau_2', & \langle \overline{\Phi}\{\lambda X.\tau_1'\}[\tau_1][\tau_2]\ f\ (\pi_1 t), \overline{\Phi}\{\lambda X.\tau_2'\}[\tau_1][\tau_2]\ f\ (\pi_2 t)\rangle \\
\text{if } u \equiv \lambda X.\tau_1' + \tau_2', & \mathbf{case}(t,\ \lambda x_1 : \tau_1'[\tau_2/X].\mathbf{inl}(\overline{\Phi}\{\lambda X.\tau_1'\}[\tau_1][\tau_2]\ f\ x_1), \\
 & \qquad\qquad \lambda x_2 : \tau_2'[\tau_2/X].\mathbf{inr}(\overline{\Phi}\{\lambda X.\tau_2'\}[\tau_1][\tau_2]\ f\ x_2)) \\
\text{if } u \equiv \lambda X.\tau_1' \rightarrow \tau_2' & \lambda x : \tau_1'[\tau_1/X]. \\
 & \qquad \overline{\Phi}\{\lambda X.\tau_2'\}[\tau_1][\tau_2]\ f\ (t(\Phi\{\lambda X.\tau_1'\}[\tau_1][\tau_2]\ f\ x)) \\
\text{if } u \equiv \lambda X.\mu(u') & \mathbf{R}\{u'[\tau_2/X]\}[\mu(u'[\tau_1/X])] \\
\text{and } Y \text{ fresh,} & \quad (\lambda x : u'[\tau_2/X][\mu(u'[\tau_1/X])] \\
 & \qquad\quad \mathbf{in}\{u'[\tau_1/X]\} \\
 & \qquad\qquad (\overline{\Phi}\{\lambda X.u'\ Y\}[\tau_1][\tau_2]\ f\ x)[\mu(u'[\tau_1/X])/Y]) \\
 & \quad t \\
\text{if } u \equiv \lambda X.\nu(u') & \mathbf{G}\{u'[\tau_1/X]\}[\nu(u'[\tau_2/X])] \\
\text{and } Y \text{ fresh,} & \quad (\lambda x : \nu(u'[\tau_2/X]). \\
 & \qquad\quad (\overline{\Phi}\{\lambda X.u'\ Y\}[\tau_1][\tau_2] \\
 & \qquad\qquad f\ (\mathbf{out}\{u'[\tau_2/X]\}x))[\nu(u'[\tau_2/X])/Y])
\end{array}$$

t

The expression $\Phi\{u\}[\tau_1][\tau_2]\ f\ t : (u\ \tau_2)$ is well-defined when

- $f : \tau_1 \rightarrow \tau_2$,
- $t : u\ \tau_1$, and
- X occurs only positively in $u\ X$, i.e., $\neg Neg(X, u\ X)$.

Intuitively, f is applied to the appropriate subterms of t, as syntactically directed by the type constructor u. Similarly, $\overline{\Phi}\{u\}[\tau_1][\tau_2]\ f\ t : (u\ \tau_1)$ is well-defined when

- $f : \tau_1 \rightarrow \tau_2$,
- $t : u\ \tau_2$, and
- X occurs only negatively in $u\ X$.

The reductions which allow the one-step cancellation of inverse constants are the significant additions to previous work. In other papers, functions have had to simulate these constants via Corollary 1. But, without these inverse cancellation reductions, reduction sequences are significantly longer.

The operational semantics is type sound:

Theorem 4 *If $\Gamma \vdash t : \sigma$ and $t \xrightarrow{*} v$, then $\Gamma \vdash v : \sigma$.*

Proof Sketch: Each evaluation step is type consistent. □

It also guarantees that evaluation terminates:

Conjecture 1 *If $\vdash t : \sigma$, then there exists a unique value v such that $t \xrightarrow{*} v$.*

Proof Sketch: By the translation given in Appendix B, all terms can be mapped to terms in F_2. The translation preserves reduction, i.e., $t \longrightarrow u$ implies $\underline{t} \xrightarrow{+}_{F_2} \underline{u}$. Since F_2 is strongly normalizing, any evaluation sequence for $\lambda^{MM\mu\nu}$ which respects the F_2 translation and standard semantics must terminate. The evaluation function $\longrightarrow$ meets this requirement. □

4 Inductive Types

Inductive types are those definable with the μ type constructor. They represent tree structures of finite depth. Some examples are

$$\begin{array}{ll}
Void & \equiv \mu(\lambda X.X) \\
Nat & \equiv \mu(\lambda X.1 + X) \\
List_A & \equiv \mu(\lambda X.1 + A \times X) \\
BinaryTree_A & \equiv \mu(\lambda X.A + X \times X)
\end{array}$$

binary tree with labels only on leaves

$AFancyTree_A \equiv \mu(\lambda X.A + A \times X \times X + A \times X \times X \times X)$

binary/ternary tree with all nodes labelled

$FancierTree_{A,B} \equiv \mu(\lambda X.A + A \times (B \rightarrow X))$

tree with B-branching and A-labelled nodes

By convention, the definition of type $(\cdot)_A$ has a free type variable A, and $(\cdot)_\tau \equiv (\cdot)_A[\tau/A]$.

While inductive types are usually defined in the form $\mu(\lambda X.\tau_1 + \cdots + \tau_n)$, this is not necessary. For example, $FancierTree_{A,B}$ is isomorphic to $\mu(\lambda X.A \times (1 + (B \rightarrow X)))$. However, any inductive type not isomorphic to $Void$ is isomorphic to some type given in the conventional form.

Observe that if X does not occur in τ, then $\mu(\lambda X.\tau)$ is isomorphic to τ.

A number of the examples in this section are adapted from [26].

4.1 Maps to inductive types (and constants)

It is helpful to first examine the structure of inductive constants and constructors. Recall that from a categorial perspective, constants are isomorphic to constructors mapping from type **1**. The patterns are most easily explained by example.

For $\mu(u) \equiv Nat$, i.e., $u \equiv \lambda X.\mathbf{1} + X$:

$0 \equiv \mathbf{in}\{u\}(\mathbf{inl}\ *)$[7]

$1 \equiv \mathbf{in}\{u\}(\mathbf{inr}\ 0)$

$2 \equiv \mathbf{in}\{u\}(\mathbf{inr}\ 1)$

$succ \equiv \lambda n : Nat.\mathbf{in}\{u\}(\mathbf{inr}\ n)$

For $\mu(u) \equiv List_A$, i.e., $u \equiv \lambda X.\mathbf{1} + A \times X$:

$null \equiv \Lambda A.\mathbf{in}\{u\}(\mathbf{inl}\ *)$[8]

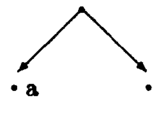
$[b] \equiv \mathbf{in}\{u\}(\mathbf{inr}\langle b, null[A]\rangle)$

$[a, b] \equiv \mathbf{in}\{u\}(\mathbf{inr}\langle a, [b]\rangle)$

$cons \equiv \Lambda A.\lambda al : A \times List_A.\mathbf{in}\{u\}(\mathbf{inr}\ al)$

For $\mu(u) \equiv BinaryTree_A$ (abbreviated BT_A), i.e., $u \equiv \lambda X.A + X \times X$:

• c $\equiv \mathbf{in}\{u\}(\mathbf{inl}\ c)$



$\equiv \mathbf{in}\{u\}(\mathbf{inr}\langle \mathbf{in}\{u\}(\mathbf{inl}\ a), \mathbf{in}\{u\}(\mathbf{inl}\ b)\rangle)$

[7] These terms are very similar to Church numerals if coproducts are encoded into the remaining calculus in the standard way. E.g., $0 \equiv \mathbf{in}\{u\}(\Lambda Z.\lambda z : Z.\lambda s : Nat \rightarrow Z.z)$ and $1 \equiv \mathbf{in}\{u\}(\Lambda Z.\lambda z : Z.\lambda s : Nat \rightarrow Z.s\ 0)$.

[8] Remember that A is free in u!

$$\begin{array}{l} \mathit{leaf} \equiv \Lambda A.\lambda a : A.\mathbf{in}\{u\}(\mathbf{inl}\ a) \\ \mathit{makeBT} \equiv \Lambda A.\lambda tt : BT_A \times BT_A.\mathbf{in}\{u\}(\mathbf{inr}\ tt) \end{array}$$

As seen from these examples and from its type, $\mathbf{in}\{u\}$ is required to "package" a term into the type $\mu(u)$.

The uncurried forms of constructors such as *cons* and *makeBT* are more natural as a result of using products in the definitions of List_A and $\mathit{BinaryTree}_A$.

4.2 Inductive functions: Maps from inductive types

When defining an inductive[9] function $g \equiv \mathbf{R}\{u\}[\tau]f$, it is often convenient to use one or both of the tools used here. One method is to give a set of recurrence equations and extract the function f. This extraction can be aided by using the commuting diagrams of F-algebras, the second method.

The form of the recurrence equations for three common inductive types (each of these having two constructors) is given here.

$$\begin{array}{ll} \mathit{Nat}: & g\ 0 \cong f_1\ * \\ & g(\mathit{succ}\ n) \cong f_2(g\ n) \\ \mathit{List}_A: & g(\mathit{null}[A]) \cong f_1\ * \\ & g(\mathit{cons}[A]\langle a, l\rangle) \cong f_2\langle a, g\ l\rangle \\ BT_A: & g(\mathit{leaf}[A]\ a) \cong f_1\ a \\ & g(\mathit{makeBT}[A]\langle t_1, t_2\rangle) \cong f_2\langle g\ t_1, g\ t_2\rangle \end{array}$$

where $f_i : \tau_i[\tau/X] \rightarrow \tau$, and $f \equiv \lambda x : u\ \tau.\mathrm{case}(x, f_1, f_2)$.

Example 2 To illustrate, we define $\mathit{even?} : \mathit{Nat} \rightarrow \mathit{Bool}$ (assume that standard Boolean functions are defined[10]). In particular, we wish to satisfy the inductive recurrences

$$\mathit{even?}\ 0 \cong \mathit{true} \qquad\qquad \mathit{even?}\ (\mathit{succ}\ n) \cong \mathit{not}(\mathit{even?}\ n)$$

The first equivalence is the same as

$$\mathit{even?}\ (0^\dagger\ *) \quad \cong \quad \mathit{true}^\dagger\ *$$

which fits the form of recurrences given at the beginning of the section. (On following examples, we leave this sort of expansion to the reader.) The above are equivalent to the commutativity of this diagram:

[9] In this context, *inductive* is equivalent to *iterative*, rather than *primitive recursive*. As will be shown in Section 4.3, iteration is as powerful as primitive recursion.

[10] These definitions are straightforward from either $\mathit{Bool} \equiv 1 + 1$ or $\mathit{Bool} \equiv \mu(\lambda X.1 + 1)$. See Appendix A.

$$\begin{tikzcd}
\mathbf{1} + Nat \arrow[r, "Id + even?"] \arrow[d, "\mathbf{in}\{\lambda N.\mathbf{1}+N\} = {[0^\dagger, succ]}"'] & \mathbf{1} + Bool \arrow[d, "f = {[true^\dagger, not]}"] \\
Nat \arrow[r, "even? = \mathbf{R}\{\lambda N.\mathbf{1}+N\}{[Bool]}f"'] & Bool
\end{tikzcd}$$

Translating these views of the desired definition into the syntax of the calculus, we first observe

$$\Phi\{\lambda N.\mathbf{1}+N\}[Nat][Bool]\ f\ t \equiv \mathbf{case}(t, \lambda u : \mathbf{1}.\mathbf{inl}\ *, \lambda n : Nat.\mathbf{inr}(f\ n))$$

and then define f using $f_1 \equiv true^\dagger$ and $f_2 \equiv not$:

$$\begin{aligned}
&f && \equiv \lambda x : \mathbf{1} + Bool.\mathbf{case}(x, true^\dagger, not)\\
&even? && \equiv \mathbf{R}\{\lambda N.\mathbf{1}+N\}[Bool]f
\end{aligned}$$

Omitting the unwieldy type information from the relevant Φ above, the following evaluation sequence demonstrates iteration using these definitions.

$$\begin{aligned}
& && even?\ 1\\
&\overset{*}{\longrightarrow} && f(\Phi\ even?\ (\mathbf{inr}\ 0))\\
&\equiv && f(\mathbf{case}(\mathbf{inr}\ 0, \lambda u : \mathbf{1}.\mathbf{inl}\ *, \lambda n : Nat.\mathbf{inr}(even?\ n)))\\
&\overset{*}{\longrightarrow} && f(\mathbf{inr}(even?\ 0))\\
&\overset{*}{\longrightarrow} && f(\mathbf{inr}(f(\Phi\ even?\ (\mathbf{inl}\ *))))\\
&\overset{*}{\longrightarrow} && f(\mathbf{inr}(f(\mathbf{inl}\ *)))\\
&\overset{*}{\longrightarrow} && not\ true\\
&\overset{*}{\longrightarrow} && false
\end{aligned}$$

Example 3 The *car*, or first element, of a list.[11]

$$\begin{aligned}
car[A](null[A]) &\cong \mathbf{inl}\ *\\
car[A](cons[A]\langle a, l\rangle) &\cong \mathbf{inr}(\pi_1\langle a, car[A]\ l\rangle)
\end{aligned}$$

[11] This definition provides "error checking", i.e., detection of traditionally erroneous $car[A](null[A])$, via coproducts. Alternatively, we could base a definition on the simpler relations

$$car[A](null[A]) \cong error \qquad\qquad car[A](cons[A]\langle a, l\rangle) \cong \pi_1\langle a, car[A]\ l\rangle$$

for some constant $error : A$.

$$\begin{tikzcd}
\mathbf{1} + A \times \mathit{List}_A \arrow[r, "\mathit{Id} + \mathit{Id} \times \mathit{car}[A]"] \arrow[d, "{[\mathit{null}[A]^{\dagger}, \mathit{cons}[A]]}"'] & \mathbf{1} + A \times (\mathbf{1} + A) \arrow[d, "\mathbf{inl} + \mathbf{inr} \circ \pi_1"] \\
\mathit{List}_A \arrow[r, "\mathit{car}[A]"'] & \mathbf{1} + A
\end{tikzcd}$$

$$\begin{aligned}
\Phi\{\lambda L.\mathbf{1} + A \times L\}\ f\ t \equiv \mathbf{case}(&t,\ \lambda u : \mathbf{1}.\mathbf{inl}\ *, \\
&\lambda al : A \times \mathit{List}_A.\mathbf{inr}\langle \pi_1\ al, f(\pi_2\ al)\rangle)
\end{aligned}$$

$$\begin{aligned}
f\ &\equiv \lambda x : \mathbf{1} + A \times (\mathbf{1} + A). \\
&\qquad \mathbf{case}(x, \lambda u : \mathbf{1}.\mathbf{inl}\ u, \lambda y : A \times (\mathbf{1} + A).\mathbf{inr}(\pi_1 y)) \\
\mathit{car}\ &\equiv \Lambda A.\mathbf{R}\{L, \mathbf{1} + A \times L\}[\mathbf{1} + A]f
\end{aligned}$$

A typical evaluation sequence of an application of *car* is

$$\begin{aligned}
& \mathit{car}[\mathit{Nat}]\ [1,2] \\
\xrightarrow{*}\ & f(\Phi\ (\mathbf{R}[\mathbf{1} + \mathit{Nat}]f)\ (\mathbf{inr}\langle 1, [2]\rangle)) \\
\equiv\ & f(\mathbf{case}(\mathbf{inr}\langle 1, [2]\rangle, \\
& \qquad \lambda u : \mathbf{1}.\mathbf{inl}\ *, \\
& \qquad \lambda al : A \times \mathit{List}_A.\mathbf{inr}\langle \pi_1\ al, \mathbf{R}[\mathbf{1} + \mathit{Nat}]f(\pi_2\ al)\rangle)) \\
\xrightarrow{*}\ & f(\mathbf{inr}\langle 1, \mathbf{R}[\mathbf{1} + \mathit{Nat}]f\ [2]\rangle) \\
\xrightarrow{*}\ & \mathbf{inr}(\pi_1\langle 1, \mathbf{R}[\mathbf{1} + \mathit{Nat}]f\ [2]\rangle) \\
\xrightarrow{*}\ & \mathbf{inr}(\pi_1\langle 1, 2\rangle) \\
\longrightarrow\ & \mathbf{inr}\ 1
\end{aligned}$$

This evaluation sequence requires a number of steps linear in the length of the list as, in the example, $\mathbf{R}[\mathbf{1} + \mathit{Nat}]f\ [2] \cong \mathit{car}[\mathit{Nat}]\ [2]$ must be evaluated. If pairing were lazy, this would not be the case, and *car* would only take constant time. However, a better definition of *car* can be given, as in Example 12, which requires only constant-time even with eager pairing.

Many inductive destructors can be defined in a similar way. The other destructors can be defined inductively as in Examples 6 and 7. In Section 4.4, we will show a much simpler way to define all of these destructors.

Example 4 Test whether all leaves in a binary tree are even numbers.

$$\mathit{leavesEven?}(\mathit{leaf}[\mathit{Nat}]\ n) \quad \cong \quad \mathit{even?}\ n$$

$$\begin{aligned} \mathit{leavesEven?}(\mathit{makeBT}[\mathit{Nat}]\langle s,t\rangle) \;\cong\;& \mathit{and}\ \langle \mathit{leavesEven?}\ s, \\ & \qquad \mathit{leavesEven?}\ t\rangle \end{aligned}$$

$$\begin{tikzcd}
\mathit{Nat} + \mathit{BT}_{\mathit{Nat}} \times \mathit{BT}_{\mathit{Nat}} \arrow[r, "\mathit{Id} + \mathit{leavesEven?}"] \arrow[d, "{[\mathit{leaf}[\mathit{Nat}], \mathit{makeBT}[\mathit{Nat}]]}"'] & \mathit{Nat} + \mathit{Bool} \times \mathit{Bool} \arrow[d, "{[\mathit{even?}, \mathit{and}]}"] \\
\mathit{BT}_{\mathit{Nat}} \arrow[r, "\mathit{leavesEven?}"'] & \mathit{Bool}
\end{tikzcd}$$

Multiple argument functions can be defined via currying as in the following example.

Example 5 Addition of two natural numbers. The recurrences

$$\mathit{plus}\ 0\ n \cong n \qquad\qquad \mathit{plus}\ (\mathit{succ}\ m)\ n \cong \mathit{plus}\ m\ (\mathit{succ}\ n)$$

are equivalent to

$$\mathit{plus}\ 0 \cong \mathit{Id} \qquad\qquad \mathit{plus}\ (\mathit{succ}\ m) \cong \lambda n : \mathit{Nat}.\mathit{plus}\ m\ (\mathit{succ}\ n)$$

The categorical equivalent of currying is exponentiation:

$$\begin{tikzcd}
\mathbf{1} + \mathit{Nat} \arrow[r, "\mathit{Id} + \mathit{Id} \times \mathit{plus}"] \arrow[d, "{[0^{\dagger}, \mathit{succ}]}"'] & \mathbf{1} + \mathit{Nat} \rightarrow \mathit{Nat} \arrow[d, "{[\mathit{Id}, \lambda y.y \circ \mathit{succ}]}"] \\
\mathit{Nat} \arrow[r, "\mathit{plus}"'] & \mathit{Nat} \rightarrow \mathit{Nat}
\end{tikzcd}$$

$$\begin{aligned} f \;&\equiv \lambda x : \mathbf{1} + \mathit{Nat} \rightarrow \mathit{Nat}. \\ &\qquad \mathrm{case}(x, \lambda u : \mathbf{1}.\mathit{Id}^{\mathit{Nat}}, \lambda y : \mathit{Nat} \rightarrow \mathit{Nat}.\lambda n : \mathit{Nat}.y(\mathit{succ}\ n)) \\ \mathit{plus} \;&\equiv \mathrm{R}[\mathit{Nat} \rightarrow \mathit{Nat}]f \end{aligned}$$

4.3 Primitive recursion

Simple induction is a valuable tool, as the previous sections shows. However, it is also overly constrained in practice. In particular, we would like to use the more convenient notion of primitive recursion such that, for example, the following recurrence equations hold.

$$\begin{array}{rl} \mathit{Nat}\colon & g\ 0 \cong f_1\ * \\ & g(\mathit{succ}\ n) \cong f_2\langle n, g\ n\rangle \\ \mathit{List}_A\colon & g(\mathit{null}[A]) \cong f_1\ * \\ & g(\mathit{cons}[A]\langle a, l\rangle) \cong f_2\langle a, \langle l, g\ l\rangle\rangle \\ \mathit{BT}_A\colon & g(\mathit{leaf}[A]\ a) \cong f_1\ a \\ & g(\mathit{makeBT}[A]\langle t_1, t_2\rangle) \cong f_2\langle\langle t_1, g\ t_1\rangle, \langle t_2, g\ t_2\rangle\rangle \end{array}$$

It is well-known that induction can implement primitive recursion, e.g., [20, 26]. However, such simulation is, an intuitive sense, frequently too inefficient. This intuition has been formalized for (the Church numeral encoding of) natural numbers by Parigot in [24]. This section shows examples of this simulation, and how to even go beyond primitive recursion. Section 4.4 shows how this inefficiency sometimes can be avoided in $\lambda^{MM\mu\nu}$.

Example 6 Natural number predecessor.[12]

$$\mathit{pred}\ 0 \cong 0 \qquad\qquad \mathit{pred}\ (\mathit{succ}\ n) \cong n$$

This is not in the form of an inductive definition, but is in the above primitive recursive form. So, since inductive types can also be encoded in pure F_2, it should not be surprising that functions such as *pred* and *cdr* may be defined using the same pairing technique common in F_2:

$$\begin{array}{rcl} \mathit{predPair}\ 0 & \cong & \langle 0, 0\rangle \\ \mathit{predPair}\ (\mathit{succ}\ n) & \cong & \langle \mathit{succ}(\pi_1(\mathit{predPair}\ n)), \pi_1(\mathit{predPair}\ n)\rangle \end{array}$$

In particular, this definition implies

$$\begin{array}{rcl} \mathit{predPair}\ n & \cong & \langle n, \mathit{pred}\ n\rangle \\ \mathit{pred}\ n & \cong & \pi_2(\mathit{predPair}\ n) \end{array}$$

$$\begin{tikzcd} 1 + \mathit{Nat} \arrow[r, "\mathit{Id} + \mathit{predPair}"] \arrow[d, "{[0^\dagger, \mathit{succ}]}"'] & 1 + \mathit{Nat} \times \mathit{Nat} \arrow[d, "f"] \\ \mathit{Nat} \arrow[r, "\mathit{predPair}"'] & \mathit{Nat} \times \mathit{Nat} \end{tikzcd}$$

[12] We omit error checking, e.g.

$$\mathit{pred}\ 0 \cong \mathbf{inl}\ * \qquad\qquad \mathit{pred}\ (\mathit{succ}\ n) \cong \mathbf{inr}\ n$$

from this and following examples for simplicity.

$$\begin{aligned}
f \quad & \equiv \lambda x : \mathbf{1} + \mathit{Nat} \times \mathit{Nat}. \\
& \qquad \mathbf{case}(x,\ \lambda u : \mathbf{1}.\langle 0, 0\rangle, \\
& \qquad\qquad \lambda nn : \mathit{Nat} \times \mathit{Nat}.\langle \mathit{succ}(\pi_1\ nn), \pi_1\ nn\rangle) \\
\mathit{predPair} & \equiv \mathbf{R}[\mathit{Nat} \times \mathit{Nat}] f \\
\mathit{pred} \quad & \equiv \lambda n : \mathit{Nat}.\pi_2(\mathit{predPair}\ n)
\end{aligned}$$

Like its F_2 counterpart, this definition requires linear time, even with alternative definitions of $\longrightarrow$, since *predPair* n must be calculated to determine *predPair*(*succ* n).

Example 7 The *cdr* of a list (such that $\mathit{cdr}[A](\mathit{null}[A]) \xrightarrow{*} \mathit{null}[A]$), using the same technique. We define

$$\begin{aligned}
\mathit{cdrPair}[A](\mathit{null}[A]) & \cong \langle \mathit{null}[A], \mathit{null}[A]\rangle \\
\mathit{cdrPair}[A](\mathit{cons}[A]\langle a, l\rangle) & \cong \langle \mathit{cons}[A]\langle a, \pi_1(\mathit{cdrPair}[A]\ l)\rangle, \pi_1(\mathit{cdrPair}[A]\ l)\rangle
\end{aligned}$$

so that

$$\begin{aligned}
\mathit{cdrPair}[A]\ l & \cong \langle l, \mathit{cdr}[A]\ l\rangle \\
\mathit{cdr}[A]\ l & \cong \pi_2(\mathit{cdrPair}[A]\ l)
\end{aligned}$$

$$\begin{tikzcd}
\mathbf{1} + A \times \mathit{List}_A \arrow[r, "\mathit{Id} + \mathit{Id} \times \mathit{cdrPair}{[A]}"] \arrow[d, "{[\mathit{null}[A]^{\dagger}, \mathit{cons}[A]]}"'] & \mathbf{1} + A \times (\mathit{List}_A \times \mathit{List}_A) \arrow[d, "f"] \\
\mathit{List}_A \arrow[r, "\mathit{cdrPair}{[A]}"'] & \mathit{List}_A \times \mathit{List}_A
\end{tikzcd}$$

$$\begin{aligned}
f \quad & \equiv \lambda x : \mathbf{1} + A \times (\mathit{List}_A \times \mathit{List}_A). \\
& \qquad \mathbf{case}(x,\ \lambda u : \mathbf{1}.\langle \mathit{null}[A], \mathit{null}[A]\rangle, \\
& \qquad\qquad \lambda y : A \times (\mathit{List}_A \times \mathit{List}_A). \\
& \qquad\qquad\quad \langle \mathit{cons}[A]\langle \pi_1 y, \pi_1(\pi_2 y)\rangle, \pi_1(\pi_2 y)\rangle) \\
\mathit{cdrPair} & \equiv \Lambda A.\mathbf{R}[\mathit{List}_A \times \mathit{List}_A] f \\
\mathit{cdr} \quad & \equiv \lambda l : \mathit{List}_A.\pi_2(\mathit{cdrPair}[A]\ l)
\end{aligned}$$

Example 8 The factorial of a natural number.

$$\mathit{fact}\ 0 \cong 1 \qquad\qquad \mathit{fact}\ (\mathit{succ}\ n) \cong \mathit{times}\ n\ (\mathit{fact}\ n)$$

Again, the above definition is not expressed in the inductive form, and we must use a pairing technique similar to before, with the relations

$$\mathit{factPair}\ 0 \cong \langle 0, 1\rangle$$

$$\begin{aligned}
factPair\ (succ\ n) &\cong \langle succ(\pi_1(factPair\ n)),\\
&\qquad times\ (succ(\pi_1(factPair\ n)))\ (\pi_2(factPair\ n))\rangle\\
factPair\ n &\cong \langle n, fact\ n\rangle\\
fact\ n &\cong \pi_2(factPair\ n)
\end{aligned}$$

The general form of primitive recursion, recurring over an inductive type $\mu(u)$, resulting in a type τ, is given by the following commuting diagram [20].

$$\begin{tikzcd}
u\ \mu(u) \arrow[d, "\mathbf{in}\{u\}"'] \arrow[r, shift right, "\Phi\{u\}\langle Id, pr\{u\}[\tau]f\rangle"'] & u(\mu(u) \times \tau) \arrow[l, shift right, "\Phi\{u\}\pi_1"'] \arrow[d, "f'"] \arrow[dr, "f"] & \\
\mu(u) \arrow[r, shift left, "\langle Id, pr\{u\}[\tau]f\rangle"] & \mu(u) \times \tau \arrow[l, shift left, "\pi_1"] \arrow[r, "\pi_2"'] & \tau
\end{tikzcd}$$

Given a function f (this corresponds to $\langle f_1, f_2\rangle$ from the beginning of the section), we get

$$f' = \langle \mathbf{in}\{u\} \circ (\Phi\{u\}\pi_1), f\rangle$$

Since the rectangle commutes,

$$\langle Id^{\mu(u)}, pr\{u\}[\tau]f\rangle = \mathbf{R}\{u\}[\mu(u) \times \tau]f'$$

So,

$$pr\{u\}[\tau]f = \pi_2 \circ (\mathbf{R}\{u\}[\mu(u) \times \tau]f')$$

$$pr\{u\} : \forall\tau.(u\ (\mu(u) \times \tau) \rightarrow \tau) \rightarrow \mu(u) \rightarrow \tau$$

Example 9 Primitive recursion for natural numbers and lists. These definitions follow the tradition of separate base- and inductive-case functions f_i, instead of a single f.

$$\begin{aligned}
&pr\{\lambda N.\mathbf{1} + N\} \equiv\\
&\quad \Lambda A.\lambda f_1 : A.\lambda f_2 : (Nat \times A) \rightarrow A.\lambda n : Nat.\\
&\qquad\qquad \pi_2(\mathbf{R}\{\lambda N.\mathbf{1} + N\}[Nat \times A]\\
&\qquad\qquad\quad (\lambda y : \mathbf{1} + Nat \times A.\\
&\qquad\qquad\qquad\quad \mathbf{case}(y,\ \lambda u : \mathbf{1}.\langle 0, f_1\rangle,\\
&\qquad\qquad\qquad\qquad\qquad \lambda na : Nat \times A.\langle succ(\pi_1\ na), f_2\ na\rangle))\\
&\qquad\qquad\quad n)
\end{aligned}$$

$$pr\{\lambda L.\mathbf{1} + A \times L\} \equiv$$

$$\begin{array}{l}
\Lambda B.\lambda f_1 : B.\lambda f_2 : A \times (List_A \times B) \rightarrow B.\lambda l : List_A. \\
\qquad\qquad \pi_2(\mathbf{R}\{L, \mathbf{1} + A \times L\}[List_A \times B] \\
\qquad\qquad\qquad (\lambda y : \mathbf{1} + A \times (List_A \times B). \\
\qquad\qquad\qquad\qquad \mathbf{case}(y,\ \lambda u : \mathbf{1}.\langle 0, f_1\rangle, \\
\qquad\qquad\qquad\qquad\qquad\qquad \lambda alb : A \times (List_A \times B). \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \langle cons[A]\langle \pi_1\ alb, \pi_1(\pi_2\ alb)\rangle, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ f_2\ alb\rangle)) \\
\qquad\qquad\qquad l)
\end{array}$$

Using these we can define, for example,

$$\begin{array}{ll}
pred & \equiv pr\{\lambda N.\mathbf{1} + N\}[Nat]\ 0\ (\lambda nn : Nat \times Nat.\pi_1\ nn), \text{ and} \\
cdr & \equiv \Lambda A.pr\{\lambda L.\mathbf{1} + A \times L\}[List_A] \\
 & \qquad\quad null[A]\ (\lambda all : A \times (List_A \times List_A).\pi_1(\pi_2\ all))
\end{array}$$

Extending this technique, we can generalize beyond the form of primitive recursion, allowing a function to depend on i previous recursive calls, for a given fixed i. I.e.,

$$\begin{array}{c}
g\ 0 \cong f_1 \qquad \ldots \qquad g(i-1) \cong f_i \\
g(i+n) \cong f_{i+1}\langle n, g\ n, \ldots, g(i+n-1)\rangle
\end{array}$$

Primitive recursion in the traditional number-theoretic sense corresponds to *prNat*[*Nat*]. However, using induction (or primitive recursion) with a higher-order type, it is possible to define functions which are not primitive recursive in the traditional sense. The common example of such a function is Ackermann's function.

Example 10 Ackermann's function.

$$\begin{array}{l}
ack \equiv \lambda m : Nat. \\
\qquad\qquad \mathbf{R}\{\lambda N.\mathbf{1} + N\}[Nat \rightarrow Nat] \\
\qquad\qquad\ (\lambda y : \mathbf{1} + Nat \rightarrow Nat. \\
\qquad\qquad\qquad \mathbf{case}(y, succ^{\dagger}, \lambda f : Nat \rightarrow Nat.\lambda n : Nat. \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{R}\{\lambda N.\mathbf{1} + N\}[Nat] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ (\lambda z : \mathbf{1} + Nat.\mathrm{case}(z, 1^{\dagger}, f)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ (succ\ n)) \\
\qquad\qquad m
\end{array}$$

4.4 Inductive destructors

Since $\mathbf{in}\{u\}$ is used to obtain the constructors for an inductive type, its inverse, $\mathbf{in}^{-1}\{u\}$, gives the corresponding destructors. For example, the destructor for *Nat* is *pred* and those for $List_A$ are *car* and *cdr*. Now observe that some destructors require linear time when using definitions such as those presented so far. Comparing these definitions to Corollary 1, we see that other definitions using $\mathbf{in}^{-1}$ are available. Due to the reduction rule using $\mathbf{in}^{-1}$, these new definitions will be more efficient.

Example 11 Using the inverse of $\mathbf{in}\{\lambda N.1 + N\}$.

$$\begin{aligned} \mathbf{in}^{-1}0 \equiv \mathbf{in}^{-1}(\mathbf{in}(\mathbf{inl}\ *)) &\longrightarrow \mathbf{inl}\ * \\ \mathbf{in}^{-1}(\mathit{succ}\ n) \equiv \mathbf{in}^{-1}(\mathbf{in}(\mathbf{inr}\ n)) &\longrightarrow \mathbf{inr}\ n \end{aligned}$$

Again defining *pred* $0 \overset{*}{\longrightarrow} 0$, we can define

$$\begin{aligned} \mathit{zero?} &\equiv \lambda n : \mathit{Nat}.\mathbf{case}(\mathbf{in}^{-1}n, \mathit{true}^\dagger, \lambda n : \mathit{Nat}.\mathit{false}) \\ \mathit{pred} &\equiv \lambda n : \mathit{Nat}.\mathbf{case}(\mathbf{in}^{-1}n, 0^\dagger, \mathit{Id}^{\mathit{Nat}}) \end{aligned}$$

And, we can compare this to Corollary 1 with the following diagram (both triangles commute) and definitions:

$$\begin{tikzcd}
1 + \mathit{Nat} \arrow[r, "\mathit{Id} + \mathbf{in}^{-1}"] \arrow[d, "\mathbf{in} = {[0^\dagger, \mathit{succ}]}"'] \arrow[dr, "\mathit{Id}"] & 1 + (1 + \mathit{Nat}) \arrow[d, "f = \mathit{Id} + \mathbf{in}"] \\
\mathit{Nat} \arrow[r, "\mathbf{in}^{-1}"'] & 1 + \mathit{Nat}
\end{tikzcd}$$

$$\begin{aligned} f &\equiv \lambda x : 1 + (1 + \mathit{Nat}).\mathbf{case}(x, \lambda u : 1.\mathbf{inl}\ u, \lambda y : 1 + \mathit{Nat}.\mathbf{inr}(\mathbf{in}\ y)) \\ \mathbf{in}^{-1} &\cong \mathbf{R}[1 + \mathit{Nat}]f \end{aligned}$$

This operational equivalence empirically confirms the corollary.

Unlike the definitions for *pred* given in the previous section, this is constant-time. For example, where n is a value,

$$\begin{aligned} & \mathit{pred}(\mathit{succ}\ n) \\ \overset{*}{\longrightarrow}\ & \mathbf{case}(\mathbf{in}^{-1}(\mathbf{in}(\mathbf{inr}\ n)), 0^\dagger, \mathit{Id}^{\mathit{Nat}}) \\ \longrightarrow\ & \mathbf{case}(\mathbf{inr}\ n, 0^\dagger, \mathit{Id}^{\mathit{Nat}}) \\ \overset{*}{\longrightarrow}\ & n \end{aligned}$$

Example 12 Using the inverse of $\mathbf{in}\{L, 1 + A \times L\}$.

$$\begin{aligned} \mathbf{in}^{-1}(\mathit{null}[A]) \equiv \mathbf{in}^{-1}(\mathbf{in}(\mathbf{inl}\ *)) &\overset{*}{\longrightarrow} \mathbf{inl}\ * \\ \mathbf{in}^{-1}(\mathit{cons}[A]\langle a, l\rangle) \equiv \mathbf{in}^{-1}(\mathbf{in}(\mathbf{inr}\langle a, l\rangle)) &\overset{*}{\longrightarrow} \mathbf{inr}\langle a, l\rangle \end{aligned}$$

$$\begin{aligned} \mathit{null?} &\equiv \Lambda A.\lambda l : \mathit{List}_A.\mathbf{case}(\mathbf{in}^{-1}l, \mathit{true}^\dagger, \lambda al : A \times \mathit{List}_A.\mathit{false}) \\ \mathit{car} &\equiv \Lambda A.\lambda l : \mathit{List}_A.\mathbf{case}(\mathbf{in}^{-1}l, \lambda u : 1.\mathbf{inl}\ *, \lambda al : A \times \mathit{List}_A.\mathbf{inr}(\pi_1\ al)) \\ \mathit{cdr} &\equiv \Lambda A.\lambda l : \mathit{List}_A.\mathbf{case}(\mathbf{in}^{-1}l, \lambda u : 1.\mathbf{inl}\ *, \lambda al : A \times \mathit{List}_A.\mathbf{inr}(\pi_2\ al)) \end{aligned}$$

5 Co-inductive Types

The dual of inductive types, co-inductive types are defined with the ν type constructor. They represent tree structures of potentially countably infinite depth. Some examples are

$\mathit{Nat\omega}$	$\equiv \nu(\lambda N.1 + N)$ natural numbers and their limit ω
Stream_A	$\equiv \nu(\lambda S.1 + A \times S)$ finite and infinite length streams
$\mathit{InfStream}_A$	$\equiv \nu(\lambda S.A \times S)$ infinite length streams
$\mathit{InfTree}_A$	$\equiv \nu(\lambda X.A \times \mathit{List}_X)$ finite branching, infinite depth trees
$\mathit{FancierITree}_{A,B}$	$\equiv \nu(\lambda X.A + A \times (B \to X))$ B-branching, potentially infinite depth trees

For each co-inductive type, there are terms of that type which represent objects of infinite size. For types such as Stream_A, there are also terms which represent finite-sized objects.

The dual types $\mu(u)$ and $\nu(u)$ represent similar collections of objects. For example, List_A and Stream_A both represent sequences of elements of type A. The co-inductive type is isomorphic to its dual inductive type plus some infinite-sized objects. Another example of this correspondence is between Nat and $\mathit{Nat\omega}$.

5.1 Co-inductive functions and simple terms: Maps to co-inductive types

For co-inductive types, there are two convenient definition methods for simple terms and constructors. The first is based on dualizing inductive simple terms and constructors. Later examples will point out a method using the categorical idea that constants of type σ are obtained from maps of type $\mathbf{1} \to \sigma$.

For $\nu(u) \equiv \mathit{Nat\omega}$, i.e., $u \equiv \lambda N.\mathbf{1} + N$:
$\mathit{zeroNat\omega} \equiv \mathbf{out}^{-1}\{u\}(\mathbf{inl}\ *)$
$\mathit{succNat\omega} \equiv \lambda n : \mathit{Nat\omega}.\mathbf{out}^{-1}\{u\}(\mathbf{inr}\ n)$
Note: ω must be defined using the second method. See Example 20.

For $\nu(u) \equiv \mathit{Stream}_A$ (abbreviated Str_A), i.e., $u \equiv \lambda S\ \mathbf{1} + A \times S$:
$\mathit{emptyStream} \equiv \Lambda A.\mathbf{out}^{-1}\{u\}(\mathbf{inl}\ *)$
$[\![a, b]\!] \equiv \mathbf{out}^{-1}\{u\}(\mathbf{inr}\langle a, \mathbf{out}^{-1}\{u\}(\mathbf{inr}\langle b, \mathit{emptyStream}[A]\rangle)\rangle)$
$\mathit{consStream} \equiv \Lambda A.\lambda \mathit{as} : A \times \mathit{Stream}_A.\mathbf{out}^{-1}\{u\}(\mathbf{inr}\ \mathit{as})$

For $\nu(u) \equiv \mathit{InfStream}_A$ (abbreviated IStr_A), i.e., $u \equiv \lambda X.A \times \mathit{List}_X$:
$\mathit{consIStream} \equiv \Lambda A.\mathbf{out}^{-1}\{u\}$

Due to duality, maps to co-inductive types are similar to maps from inductive types, in that such functions involve the induction operator and commuting diagrams.

However, the general recurrence patterns of co-induction are not as convenient as those of induction as shown in Section 4.2. To define a function $g \equiv \mathbf{G}[X]f : X \rightarrow \nu(\lambda X.\tau)$, the recurrence equations of co-induction for the above three types are

$$\begin{aligned}
&\mathit{Nat\omega}\text{:}\ \ g\ x \cong \mathbf{case}(f\ x, \mathit{zeroNat\omega}^{\dagger}, \lambda y : X.\mathit{succNat\omega}(g\ y))\\
&\mathit{Str}_A\text{:}\ \ g\ x \cong \mathbf{case}(f\ x,\ \mathit{emptyStream}[A]^{\dagger},\\
&\qquad\qquad\qquad\qquad \lambda ax : A \times X.\mathit{consStream}[A]\langle \pi_1\ ax, g(\pi_2\ ax)\rangle)\\
&\mathit{IStr}_A\text{:}\ \ g\ x \cong \mathit{consIStream}[A]\langle \pi_1(f\ x), g(\pi_2(f\ x))\rangle
\end{aligned}$$

Example 13 The empty stream (alternate method). Using the idea that the constant is essentially a map of type $\mathbf{1} \rightarrow \mathit{Stream}_A$, we use the general pattern of co-induction on streams.

$$\begin{tikzcd}
\mathbf{1} + A \times \mathbf{1} \arrow[r, "\mathit{Id} + \mathit{Id} \times \mathit{emptyStream}{[A]}^{\dagger}"] & \mathbf{1} + A \times \mathit{Str}_A \\
\mathbf{1} \arrow[u, "f = \mathbf{inl}"] \arrow[r, "\mathit{emptyStream}{[A]}^{\dagger}"'] & \mathit{Str}_A \arrow[u, "\mathbf{out}"']
\end{tikzcd}$$

$$\begin{aligned}
\Phi\{\lambda S.A \times S\}[\mathbf{1}][\mathit{Str}_A]\ f\ t \equiv\ & \mathbf{case}(t,\ \lambda u : \mathbf{1}.\mathbf{inl}\ *,\\
& \lambda au : A \times \mathbf{1}.\mathbf{inr}\langle \pi_1\ au, f(\pi_2\ au)\rangle)
\end{aligned}$$

$$\mathit{emptyStream} \equiv \Lambda A.\mathbf{G}\{\lambda S.\mathbf{1} + A \times S\}[\mathbf{1}](\lambda u : \mathbf{1}.\mathbf{inl}\ u)*$$

Evaluation of $\mathbf{out}(\mathit{emptyStream}[A])$:

$$\begin{aligned}
& \mathbf{out}(\mathit{emptyStream}[A])\\
\longrightarrow\ & \Phi\ (\mathbf{G}[\mathbf{1}](\lambda u : \mathbf{1}.\mathbf{inl}\ u))\ ((\lambda u : \mathbf{1}.\mathbf{inl}\ u)\ *)\\
\equiv\ & \mathbf{case}((\lambda u : \mathbf{1}.\mathbf{inl}\ u)*,\\
& \quad \lambda u : \mathbf{1}.\mathbf{inl}\ *,\\
& \quad \lambda au : A \times \mathbf{1}.\mathbf{inr}(\langle \pi_1\ au, \mathbf{G}[\mathbf{1}](\lambda u : \mathbf{1}.\mathbf{inl}\ u)(\pi_2\ au)\rangle))\\
\xrightarrow{*}\ & \mathbf{inl}\ *
\end{aligned}$$

Example 14 Similarly, we can define *consIStream* using this alternate method, and compare the result to Corollary 1. It fits into the above recurrence pattern as

$$\mathit{consIStream}[A]\langle a, s\rangle \cong \mathit{consIstream}[A]\langle a, \mathit{consIStream}[A](\mathbf{out}\ s)\rangle$$

since $\mathit{consIStream}[A] \cong \mathbf{out}^{-1}$.

$$\begin{tikzcd}
A \times (A \times \mathit{IStr}_A) \arrow[r, "Id \times \mathit{consIStream}[A]"] & A \times \mathit{IStr}_A \\
A \times \mathit{IStr}_A \arrow[u, "Id \times \mathbf{out}"] \arrow[r, "\mathit{consIStream}[A]"'] & \mathit{IStr}_A \arrow[u, "\mathbf{out}"']
\end{tikzcd}$$

$$\Phi\{\lambda S.A \times S\}[A \times \mathit{IStr}_A][\mathit{IStr}_A]\ f\ t \equiv \langle \pi_1 t, f(\pi_2 t)\rangle$$

$$\begin{aligned}
f &\equiv \lambda as : A \times \mathit{IStr}_A.\langle \pi_1\ as, \mathbf{out}(\pi_2\ as)\rangle \\
\mathit{consIStream} &\equiv \Lambda A.\mathbf{G}[A \times \mathit{IStr}_A]f
\end{aligned}$$

Unfortunately, destructing a term built using this definition is computationally expensive. In essense, destructing a stream built with this *consIStream* propagates the **out** found in the above definition of f. E.g., given any $is : \mathit{IStream}_{Nat}$,

$$\begin{aligned}
& \mathit{tailIStr}[A](\mathit{consIStream}[A]\langle 1, is\rangle) \\
\xrightarrow{*}\ & \pi_2(\mathbf{out}(\mathit{consIStream}[A]\langle 1, is\rangle)) \\
\xrightarrow{*}\ & \pi_2(\Phi\ (\mathbf{G}[A \times \mathit{IStr}_A]f)\ (f\langle 1, is\rangle)) \\
\equiv\ & \pi_2(\langle \pi_1(f\langle 1, is\rangle), \mathbf{G}[A \times \mathit{IStr}_A]f(\pi_2(f\langle 1, is\rangle))\rangle) \\
\xrightarrow{*}\ & \mathbf{G}[A \times \mathit{IStr}_A]f(\mathbf{out}\ is)
\end{aligned}$$

The last term in this reduction sequence is observationally equivalent to *is*, but computing the *headIStr* or *tailIStr* of this stream requires a longer reduction sequence than does destructing *is*. Using the previous definition of $\mathit{consIStream} \equiv \mathbf{out}^{-1}$ avoids this problem since

$$\pi_2(\mathit{tailIStr}[A](\mathbf{out}^{-1}\langle 1, is\rangle)) \xrightarrow{*} is.$$

So, just as the inverse of $\mathbf{in}\{u\}$ efficiently defines an inductive type's destructors, the inverse of $\mathbf{out}\{u\}$ produces efficient co-inductive constructors.

Example 15 The infinite stream of natural numbers from a given one, e.g., *natsFrom* $2 \cong [\![2, 3, 4, \ldots]\!]$. From the relation

$$\mathit{natsFrom}\ n \cong \mathit{consIStream}[\mathit{Nat}]\langle n, \mathit{natsFrom}(\mathit{succ}\ n)\rangle$$

we can obtain the equivalent

$$\mathbf{out}\ (\mathit{natsFrom}\ n) \cong \langle n, \mathit{natsFrom}\ (\mathit{succ}\ n)\rangle$$

which is the natural counterpart for the commuting diagram

$$\begin{tikzcd}
\mathit{Nat} \times \mathit{Nat} \arrow[r, "\mathit{Id} \times \mathit{natsFrom}"] & \mathit{Nat} \times \mathit{IStr}_{\mathit{Nat}} \\
\mathit{Nat} \arrow[u, "\langle \mathit{Id}, \mathit{succ}\rangle"] \arrow[r, "\mathit{natsFrom}"'] & \mathit{IStr}_{\mathit{Nat}} \arrow[u, "\mathbf{out}"']
\end{tikzcd}$$

$$\mathit{natsFrom} \equiv \mathrm{G}[\mathit{Nat}](\lambda n : \mathit{Nat}.\langle n, \mathit{succ}\ n\rangle)$$

Example 16 The infinite stream of a constant $c : A$. The recurrence

$$\mathit{constIS}[A]c \cong \mathit{consIStream}[A]\langle c, \mathit{constIS}[A]c\rangle$$

leads to the definition

$$\mathit{constIS} \equiv \Lambda A.\lambda c : A.\mathrm{G}\{\lambda S.A \times S\}[\mathbf{1}](\lambda u : \mathbf{1}.\langle c, u\rangle)\ *$$

Example 17 The infinite stream of factorial numbers. We first define

$$\mathit{factsHelp}\ \langle m, n\rangle \cong \mathit{consIStream}[\mathit{Nat}]\langle n, \mathit{factsHelp}\ \langle \mathit{times}\ m\ n, \mathit{succ}\ n\rangle\rangle$$

which encapsulates the incremental computation of the factorials. To define the stream of all factorials, we must seed the computation with some numbers, in this case the first factorial number and the first number to multiply by:

$$\mathit{facts} \equiv \mathit{factsHelp}\ \langle 1, 1\rangle$$

$$\begin{tikzcd}
\mathit{Nat} \times (\mathit{Nat} \times \mathit{Nat}) \arrow[r, "\mathit{Id} \times \mathit{factsHelp}"] & \mathit{Nat} \times \mathit{IStr}_{\mathit{Nat}} \\
\mathit{Nat} \times \mathit{Nat} \arrow[u, "f"] \arrow[r, "\mathit{factsHelp}"'] & \mathit{IStr}_{\mathit{Nat}} \arrow[u, "\mathbf{out}"']
\end{tikzcd}$$

$$\begin{aligned}
f &\equiv \lambda nn : \mathit{Nat} \times \mathit{Nat}. \\
&\qquad\qquad \langle \pi_1\ nn, \langle \mathit{times}\ (\pi_1\ nn)\ (\pi_2\ nn), \mathit{succ}(\pi_2\ nn)\rangle\rangle \\
\mathit{factsHelp} &\equiv \mathbf{G}[\mathit{Nat} \times \mathit{Nat}]f \\
\mathit{facts} &\equiv \mathit{factsHelp}\ \langle 1, 1\rangle
\end{aligned}$$

Example 18 The infinite stream of Fibonacci numbers

$$\mathit{fibsHelp}\ \langle m, n\rangle) \cong \mathit{consIStream}[\mathit{Nat}]\langle m, \mathit{fibsHelp}\ \langle n, \mathit{plus}\ m\ n\rangle\rangle$$

$$\begin{aligned}
f &\equiv \lambda nn : \mathit{Nat} \times \mathit{Nat}.\langle \pi_1\ nn, \langle \pi_2\ nn, \mathit{plus}\ (\pi_1\ nn)\ (\pi_2\ nn)\rangle\rangle \\
\mathit{fibsHelp} &\equiv \mathbf{G}[\mathit{Nat} \times \mathit{Nat}]f \\
\mathit{fibs} &\equiv \mathit{fibsHelp}\ \langle 0, 1\rangle
\end{aligned}$$

Example 19 Appending two streams. The following definition is based on the relations

$$\begin{aligned}
\mathit{appStr}[A]\langle e, e\rangle &\cong e \\
\mathit{appStr}[A]\langle e, \mathit{consStr}[A]\langle a, s\rangle\rangle &\cong \mathit{consStr}[A]\langle a, \mathit{appStr}[A]\langle e, s\rangle\rangle \\
\mathit{appStr}[A]\langle \mathit{consStr}[A]\langle a, s\rangle, t\rangle &\cong \mathit{consStr}[A]\langle a, \mathit{appStr}[A]\langle s, t\rangle\rangle
\end{aligned}$$

using the abbreviation $e \equiv \mathit{emptyStream}[A]$.

Using destructors defined in Section 5.3 and boolean functions defined in Appendix A,

$$\begin{aligned}
f &\equiv \lambda ss : \mathit{Str}_A \times \mathit{Str}_A. \\
&\qquad\qquad \mathit{ite}\ (\mathit{emptyStream}?[A](\pi_1\ ss)) \\
&\qquad\qquad\quad \langle \mathit{ite}\ (\mathit{emptyStream}?[A](\pi_2\ ss)) \\
&\qquad\qquad\qquad \langle \mathbf{inl}\ *, \\
&\qquad\qquad\qquad\ \mathbf{inr}\langle \mathit{headStr}[A](\pi_2\ ss), \langle \mathbf{inl}\ *, \mathit{tailStr}[A](\pi_2\ ss)\rangle\rangle\rangle \\
&\qquad\qquad\quad\ \mathbf{inr}\langle \mathit{headStr}[A](\pi_1\ ss), \langle \mathit{tailStr}[A](\pi_1\ ss), \pi_2\ ss\rangle\rangle\rangle \\
\mathit{appStr} &\equiv \Lambda A.\mathbf{G}[\mathit{Str}_A \times \mathit{Str}_A]f
\end{aligned}$$

Example 20 The term $\omega : \mathit{Nat\omega}$. The desired equality $\omega \cong \mathit{succNat\omega}\ \omega$ is equivalent to **out** $(\omega^\dagger\ *) \cong \mathbf{inr}(\omega^\dagger\ *)$. So, we have $\omega \equiv \mathbf{G}[\mathbf{1}](\lambda u : \mathbf{1}.\mathbf{inr}\ u)\ *$. Note than an alternate definition for $\mathit{zeroNat\omega}$ is the very similar $\mathbf{G}[\mathbf{1}](\lambda u : \mathbf{1}.\mathbf{inl}\ u)\ *$.

Since $\mathit{Nat\omega}$ is isomorphic to $\mathit{Stream}_{\mathbf{1}}$, we can adapt Example 19 to define (non-curried) addition on $\mathit{Nat\omega}$:

$$\begin{aligned}
f &\equiv \lambda nn : \mathit{Nat\omega} \times \mathit{Nat\omega}. \\
&\qquad\qquad \mathit{ite}\ (\mathit{zeroNat\omega}?(\pi_1\ nn)) \\
&\qquad\qquad\quad \langle \mathit{ite}\ (\mathit{zeroNat\omega}?(\pi_2\ nn)) \\
&\qquad\qquad\qquad \langle \mathbf{inl}\ *, \mathbf{inr}\langle \mathbf{inl}\ *, \mathit{predNat\omega}(\pi_2\ nn)\rangle\rangle \\
&\qquad\qquad\quad\ \mathbf{inr}\langle \mathit{predNat\omega}(\pi_1\ nn), \pi_2\ nn\rangle\rangle \\
\mathit{plusNat\omega} &\equiv \mathbf{G}[\mathit{Nat\omega} \times \mathit{Nat\omega}]f
\end{aligned}$$

5.2 Primitive co-recursion

Dualizing the diagram for primitive recursion, we obtain the following commuting diagram [20].

$$\begin{tikzcd}
 & u\ (\nu(u)+\tau) \arrow[r, shift right, "{\Phi\{u\}\ [Id, pcor\{u\}[\tau]f]}"'] & u\ \nu(u) \arrow[l, shift right, "\Phi\{u\}\ \mathbf{inl}"'] \\
\tau \arrow[r, "\mathbf{inr}"'] \arrow[ur, "f"] & \nu(u)+\tau \arrow[u, "f'"] \arrow[r, shift left, "{[Id, pcor\{u\}[\tau]f]}"] & \nu(u) \arrow[u, "\mathbf{out}\{u\}"'] \arrow[l, shift left, "\mathbf{inl}"]
\end{tikzcd}$$

So, given a function f, we find f' by

$$f' = [(\Phi\{u\}\ \mathbf{inl}) \circ \mathbf{out}\{u\}, f]$$

Then, since the rectangle commutes,

$$[Id, pcor\{u\}[\tau]f] = \mathbf{G}\{u\}[\nu(u)+\tau]f'$$

So,

$$pcor\{u\}[\tau]f = \mathbf{inr} \circ \mathbf{G}\{u\}[\nu(u)+\tau]f'$$

$$pcor\{u\} : \forall\tau.(\tau \rightarrow u\ (\nu(u)+\tau)) \rightarrow \tau \rightarrow \nu(u)$$

Recurrence equations for a primitive co-recursive function g would be based on the equation

$$g\ x \cong \Phi\{u\}\ (\lambda y : \nu(u) + X.\mathbf{case}(y, Id^{\nu(u)}, g))\ (f\ x)$$

for each constructor u.

Example 21 Translating the definition of *pcor* into $\lambda^{MM\mu\nu}$ syntax leads to, for example,

$$\begin{array}{l}
pcor\{Nat\omega\} \equiv \Lambda A.\lambda f : A \rightarrow \mathbf{1} + (Nat\omega + A).\lambda x : A. \\
\qquad\qquad\qquad \mathbf{inr}(\mathbf{G}\{Nat\omega\}[Nat\omega + A] \\
\qquad\qquad\qquad\qquad (\lambda y : Nat\omega + A. \\
\qquad\qquad\qquad\qquad\quad \mathbf{case}(y,\ \lambda n : Nat\omega. \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{case}(\mathbf{out}\{\lambda N.\mathbf{1}+N\}n, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \lambda u : \mathbf{1}.\mathbf{inl}\ *, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \lambda m : Nat\omega.\mathbf{inr}(\mathbf{inl}\ m)), \\
\qquad\qquad\qquad\qquad\qquad\qquad f)) \\
\qquad\qquad\qquad\qquad x)
\end{array}$$

An example of *pcor* is the *zip* function, which maps a pair of streams into a stream of pairs.

$$\begin{aligned}
\mathit{zipIS} \equiv \Lambda A.\mathit{pcor}\{\lambda S.A \times S\}[\mathit{IS}_A \times \mathit{IS}_A] \\
(\lambda ss : \mathit{IS}_A \times \mathit{IS}_A. \\
\langle\langle \mathbf{\mathit{headIStr}}[A](\pi_1\ ss), \mathbf{\mathit{headIStr}}[A](\pi_2\ ss)\rangle \\
\mathbf{inr}\langle \mathit{tailIStr}[A](\pi_1\ ss), \mathit{tailIStr}[A](\pi_2\ ss)\rangle\rangle)
\end{aligned}$$

Notice the similarity in structure to the definitions of *appStr* and *plusNatω* which could also be defined via primitive co-recursion.

5.3 Maps from co-inductive types

The use of $\mathbf{out}\{u\}$ is characteristic of such functions from co-inductive types, as it "unrolls" or "forces" an object one step, the only way to access the information packaged by $\mathbf{G}\{u\}$. The simplest examples of its use are destructors and base-case tests.

Example 22 Test for the empty stream.

$$\mathit{emptyStream?} \equiv \Lambda A.\lambda s : \mathit{Str}_A.\mathbf{case}(\mathbf{out}\ s, \mathit{true}^{\dagger}, \lambda as : A \times \mathit{Str}_A.\mathit{false})$$

Example 23 The head and tail destructors for streams.

$$\begin{aligned}
\mathit{headStr} &\equiv \Lambda A.\lambda s : \mathit{Str}_A.\mathbf{case}(\mathbf{out}\ s,\ \lambda u : \mathbf{1}.\mathbf{inl}\ *, \\
&\qquad\qquad\qquad\qquad\qquad\quad \lambda as : A \times \mathit{Str}_A.\mathbf{inr}(\pi_1\ as)) \\
\mathit{tailStr} &\equiv \Lambda A.\lambda s : \mathit{Str}_A.\mathbf{case}(\mathbf{out}\ s,\ \lambda u : \mathbf{1}.\mathbf{inl}\ *, \\
&\qquad\qquad\qquad\qquad\qquad\quad \lambda as : A \times \mathit{Str}_A.\mathbf{inr}(\pi_2\ as))
\end{aligned}$$

$$\begin{aligned}
\mathit{headIStr} &\equiv \Lambda A.\lambda s : \mathit{IStr}_A.\pi_1(\mathbf{out}\ s) \\
\mathit{tailIStr} &\equiv \Lambda A.\lambda s : \mathit{IStr}_A.\pi_2(\mathbf{out}\ s)
\end{aligned}$$

Example 24 Maps from *Natω*.

$$\begin{aligned}
\mathit{zeroNat}\omega? &\equiv \lambda n : \mathit{Nat}\omega.\mathbf{case}(\mathbf{out}\ n, \mathit{true}^{\dagger}, \lambda m : \mathit{Nat}\omega.\mathit{false}) \\
\mathit{predNat}\omega &\equiv \lambda n : \mathit{Nat}\omega.\mathbf{case}(\mathbf{out}\ n, \mathit{zero}^{\dagger}, \lambda m : \mathit{Nat}\omega.m)
\end{aligned}$$

As desired, $\mathit{predNat}\omega\ \omega \xrightarrow{*} \omega$.

The function $\omega?$ cannot be defined. Since $\mathbf{out}^{-1}$ provides the only way to "unroll" and examine an object of type *Natω*, the only way to define such as test is by the above destructors. Thus, such a function would have to decrement its argument until *zeroNatω* was obtained, so it cannot be written in the calculus. Similarly, any such test of infiniteness on any co-inductive type is impossible.

Example 25 The function ***nextnode*** returns the next node of the tree paired with a new tree with which to continue the search.

$$\begin{aligned}
\mathit{nextnode} \equiv \Lambda A.\lambda t :\mathit{InfTree}_A.\ & \\
& \langle \pi_1(\mathbf{out}\ t), \\
& \quad \mathit{makeIT}_A\langle \pi_1(\mathbf{out}(\mathit{car}[A](\pi_2(\mathbf{out}\ t)))), \\
& \qquad\qquad \mathit{append}[\mathit{InfTree}_A] \\
& \qquad\qquad\qquad (\mathit{cdr}[A](\pi_2(\mathbf{out}\ t)), \\
& \qquad\qquad\qquad\ \pi_2(\mathbf{out}(\mathit{car}[A](\pi_2(\mathbf{out}\ t))))))\rangle\rangle
\end{aligned}$$

where $\mathit{makeIT}_A \equiv \mathbf{out}^{-1}$ is the constructor for $\mathit{InfTree}_A$.

Repeated uses of *nextnode* results in breadth-first search of a finitely branching infinite tree:

$$\mathit{search}[A]t \cong \mathit{consiS}[A]\langle \pi_1(\mathit{nextnode}[A]t), \mathit{search}[A](\mathit{nextnode}[A]t)\rangle$$

$$\mathit{search} \equiv \Lambda A.\mathbf{G}\{\lambda S.A \times S\}[\mathit{IT}_A](\mathit{nextnode}[A])$$

Thus, *search*$[A]t$ is an infinite stream of the nodes of t in breadth-first order.

6 Summary of Related Work

Hagino [8, 9] uses a generalization of algebras called *di-algebras*. This allows him not to assume any base types or the constructors + and ×. Instead, all types are defined with a μ or ν constructor and a possibly empty list of functors corresponding to the τ_i's in $\mu(\lambda X.\tau_1 + \cdots + \tau_n)$.

Coquand and Paulin [5], and Pfenning and Paulin-Mohring [25] are similar in that they also do not assume any base types or the constructors + and ×. However, they do not work in a category theoretic framework and they use inductive, but not co-inductive types.

Mendler [20] explains primitive recursion and its dual in terms of category theory, using a generalization of algebras. Using these as primitives instead of (co-)induction, a calculus using the ideas outlined would allow constant-time encodings of our inverses.

Pierce, Dietzen, and Michaylov [26] present an example-based tutorial on programming in the F_i hierarchy of calculi, using iterators for inductive types.

Both Leivant [12] and Parigot [22, 23, 24] view programs with inductive data types as being derived from proofs. Leivant formalizes the extraction of programs from several families of calculi, giving numerous examples. Instead of extending a calculus to improve efficiency, Parigot examines alternate encodings of *Nat* in F_2 (optionally extended with *fix*) which allow constant-time destructors.

Michaylov and Pfenning [21] describe a process to compile F_2 terms to F_2 extended with constants for inductive constructors and recursors. It translates, for example, the common pair-based *pred* function similar to Example 6 to a constant-time function using recursors. A more systematic approach to defining the extensions in their target calculus could be obtained from our μ types and related terms.

Leivant [13] looks at Church numerals in a predicative version of F_2, describing precisely what computations can be defined on that type. When adding inductive types to this stratified calculus, he shows that the type $\mu X.\tau$ is at the same level as τ. In $\lambda^{MM\mu\nu}$, this means than $\mu(\lambda X.\tau)$ is a type and not a type scheme. He also proves that the addition of inductive types "does not result in new functions being representable, but it does allow new *algorithms*".

Burstall [4] extends ML with an inductive case statement, and relates programming with inductive types to specification with abstract data types, another area which uses initial algebras and sometimes final co-algebras. Hagino [10] extends ML with a "codatatype" declaration and a "merge" statement which corresponds to G. Wraith [27] uses a rougher equivalent system. Both notations, however, assume definitions of co-inductive types are of the form $\nu(\lambda X.\tau_1 \times \cdots \times \tau_n)$, which makes use of types such as $Stream_A$ inconvenient.

Crole [6] gives a model for an inductive calculus.

With the assumption that $\mathcal{C}$ is the category of CPO's, Meijer, Fokkinga, and Paterson [15, 7] eliminate the distinction between least and greatest fixed points. Thus, $\mathbf{in}^{-1}\{u\} = \mathbf{out}\{u\}$. This allows additional elegant recursion schemes, but introduces non-strictness.

7 Conclusions

Algebraic datatypes are a valuable abstraction for programming, as terms are easily defined directly from their specifications, i.e., recurrence equations or simple category theory diagrams. Using the morphisms $\mathbf{in}^{-1}$ and $\mathbf{out}^{-1}$ provides straightforward means of obtaining constant-time inductive destructors and co-inductive constructors, which significantly improves efficiency as compared to similar calculi. It has also been shown that conceptually infinite objects can be used with ease. However, when termination is guaranteed, the usefulness of co-inductive datatypes is significantly restricted, as many common functions cannot be defined.

8 Comments and Future Research

The calculus as presented is rather verbose from explicit types. Type inference should be explored to eliminate or reduce the amount of explicit type information necessary. A ML-like type declaration facility together with pattern matching, as in [4, 27, 10] would also be useful, but work still needs to be done for co-inductive types.

A formal model of the calculus would involve formalizing the points raised in Section 2, in particular, detailing the structure of the "category of all types".

It would be interesting to base the calculus on the full λ^{ML} calculus, reintroducing higher kinds. In that case, it is more difficult to enforce the positivity constraint on (co-)inductive types that ensures that Φ is well-defined. Alternatively, dropping the positivity constraint altogether introduces non-termination and requires redefinition of the evaluation of **R** and **G**, since Φ is not always definable.[14]

The syntax of the calculus creates one unfortunate semantic problem. It is not Church-Rosser when using the standard η rule and an inductive η-like rule corresponding to Theorem 2, $\mathbf{R}\{u\}[\mu(u)]\mathbf{in}\{u\}\ t \longleftrightarrow t$. In this case, the diamond property does not hold for $\lambda x : \mu(u).\mathbf{R}\{u\}[\mu(u)]\mathbf{in}\{u\}\ t$. A simple fix is to only allow "fully applied" forms such as $\mathbf{R}\{u\}[\tau]f\ t$ to be terms. The problem does not seem to arise with the alternate inductive η-like rule, $\mathbf{R}\{u\}[\mu(u)]\mathbf{in}\{u\} \longleftrightarrow Id^{\mu(u)}$.

More could be learned about (co-)inductive terms in F_2 by translating our examples, for example, as shown in Appendix B. Also, we would like to examine more closely the duality of types $\mu(u)$ and $\nu(u)$. Furthermore, our familiarity of co-inductive constructs is still not as developed as our understanding of programming with inductive types. More examples could come from extracting co-inductive programs from proofs.

Acknowledgements

Many thanks to Chris Colby, Daniel Leivant, the ever-questioning Mark Lillibridge, and Benjamin Pierce, and especially Bob Harper and Frank Pfenning for their ideas, suggestions, and proof-reading.

A Other functions

The most direct definition of *Bool*, with some typical functions:

$$\begin{array}{ll}
Bool & \equiv \mathbf{1}+\mathbf{1} \\
true & \equiv \mathbf{inl}\ * \\
and & \equiv \lambda bb : Bool \times Bool.\mathbf{case}(\pi_1\ bb, \pi_2\ bb, false) \\
ite & \equiv \lambda b : Bool.\lambda aa : A \times A.\mathbf{case}(b, \lambda u : \mathbf{1}.\pi_1\ aa, \lambda u : \mathbf{1}.\pi_2\ aa)
\end{array}$$

An alternative definition, allowing use of the iterator **R** to define *ite*:

$$\begin{array}{ll}
Bool & \equiv \mu(\lambda X.\mathbf{1}+\mathbf{1}) \\
true & \equiv \mathbf{in}(\mathbf{inl}\ *) \\
and & \equiv \lambda bb : Bool \times Bool.\mathbf{case}(\pi_1(\mathbf{in}^{-1} bb), \pi_2(\mathbf{in}^{-1} bb), false) \\
ite & \equiv \lambda b : Bool.\mathbf{R}[A \times A \rightarrow A](\lambda x : \mathbf{1}+\mathbf{1}.\mathbf{case}(x, \lambda u : \mathbf{1}.\lambda aa : A \times A.\pi_1\ aa, \\
 & \qquad \lambda u : \mathbf{1}.\lambda aa : A \times A.\pi_2\ aa))
\end{array}$$

Some other basic functions on natural numbers; note the similarity of *monus* and *plus* (Exercise 5):

$$\begin{array}{ll}
\mathit{eq?} & \equiv \mathbf{R}[\mathit{Nat} \to \mathit{Bool}] \\
& \quad (\lambda x :1 + \mathit{Nat} \to \mathit{Bool}. \\
& \qquad \mathbf{case}(x, \mathit{zero?}^{\dagger}, \lambda y : \mathit{Nat} \to \mathit{Bool}.\lambda n : \mathit{Nat}.y(\mathit{pred}\ m)) \\
\mathit{leq?} & \equiv \mathbf{R}[\mathit{Nat} \to \mathit{Bool}] \\
& \quad (\lambda x : 1 + \mathit{Nat} \to \mathit{Bool}. \\
& \qquad \mathbf{case}(x,\ \lambda u : \mathbf{1}.\lambda n : \mathit{Nat}.\mathit{true}, \\
& \qquad\qquad\quad \lambda y : \mathit{Nat} \to \mathit{Bool}.\lambda n : \mathit{Nat}. \\
& \qquad\qquad\qquad \mathit{ite}\ (\mathit{zero?}\ n)\ \langle \mathit{false}, y(\mathit{pred}\ n)\rangle)))
\end{array}$$

$$\begin{array}{ll}
\mathit{monusHelp} & \equiv \mathbf{R}[\mathit{Nat} \to \mathit{Nat}] \\
& \quad (\lambda x : 1 + \mathit{Nat} \to \mathit{Nat}. \\
& \qquad \mathbf{case}(x,\ \lambda u : \mathbf{1}.\mathit{Id}^{\mathit{Nat}}, \\
& \qquad\qquad\quad \lambda y : \mathit{Nat} \to \mathit{Nat}.\lambda n : \mathit{Nat}.y(\mathit{pred}\ m)) \\
\mathit{monus} & \equiv \lambda n : \mathit{Nat}.\lambda m : \mathit{Nat}.\mathit{monusHelp}\ m\ n
\end{array}$$

$$\begin{array}{ll}
\mathit{divrem} & \equiv \mathbf{R}[\mathit{Nat} \to (\mathit{Nat} \times \mathit{Nat})] \\
& \quad (\lambda x : 1 + \mathit{Nat} \to (\mathit{Nat} \times \mathit{Nat}). \\
& \qquad \mathbf{case}(x,\ \lambda u : \mathbf{1}.\lambda m : \mathit{Nat}.\langle 0, 0\rangle, \\
& \qquad\qquad\quad \lambda y : \mathit{Nat} \to (\mathit{Nat} \times \mathit{Nat}).\lambda n : \mathit{Nat}. \\
& \qquad\qquad\qquad \mathit{ite}\ (\mathit{eq?}\ (\pi_2(y\ n))\ (\mathit{pred}\ n)) \\
& \qquad\qquad\qquad\quad \langle \mathit{succ}(\pi_1(y\ n)), 0\rangle \\
& \qquad\qquad\qquad\quad \langle \pi_1(y\ n), \mathit{succ}(\pi_2(yn))\rangle)) \\
\mathit{div} & \equiv \lambda m : \mathit{Nat}.\lambda n : \mathit{Nat}.\pi_1(\mathit{divrem}\ m\ n) \\
\mathit{rem} & \equiv \lambda n : \mathit{Nat}.\lambda m : \mathit{Nat}.\pi_2(\mathit{divrem}\ m\ n)
\end{array}$$

$$\begin{array}{ll}
\mathit{diff} & \equiv \mathit{prNat}[\mathit{Nat} \to \mathit{Nat}]\ \mathit{Id}^{\mathit{Nat}} \\
& \qquad\qquad\qquad (\lambda x : \mathit{Nat} \times (\mathit{Nat} \to \mathit{Nat}).\lambda n : \mathit{Nat}. \\
& \qquad\qquad\qquad\quad \mathit{ite}\ (\mathit{zero?}\ n)\ \langle \mathit{succ}(\pi_1 x), (\pi_2 x)\ (\mathit{pred}\ n)\rangle)
\end{array}$$

Filtering and accumulating, on a list:

$$\begin{array}{ll}
\mathit{filter} & \equiv \lambda p : A \to \mathit{Bool}. \\
& \qquad \mathbf{R}[\mathit{List}_A](\lambda x : 1 + A \times \mathit{List}_A. \\
& \qquad\qquad\qquad \mathbf{case}(x,\ \lambda u : \mathbf{1}.\mathbf{inl}\ *, \\
& \qquad\qquad\qquad\qquad\quad \lambda al : A \times \mathit{List}_A.\mathit{ite}\ (p(\pi_1\ al))\ \langle al, \pi_2\ al\rangle)) \\
\mathit{accum} & \equiv \mathbf{R}[((A \times B) \to B) \to B \to B] \\
& \quad (\lambda x : 1 + A \times (((A \times B) \to B) \to B \to B). \\
& \qquad \mathbf{case}(x,\ \lambda u : \mathbf{1}.\lambda f : (A \times B) \to B.\mathit{Id}^{B}, \\
& \qquad\qquad\quad \lambda y : A \times (((A \times B) \to B) \to B \to B). \\
& \qquad\qquad\qquad \lambda f : (A \times B) \to B.\lambda b : B. \\
& \qquad\qquad\qquad\quad (\pi_2 y)\ f\ (f\langle \pi_1 y, b\rangle)))
\end{array}$$

The equivalent functions on streams are not total and, therefore, not definable in the calculus.

Merging sorted infinite streams, allowing duplicates:

$$\begin{aligned}
&\mathit{mergeIS} \equiv \Lambda A.\mathbf{G}[IS_A \times IS_A]\\
&\qquad\qquad (\lambda ss : IS_A \times IS_A.\\
&\qquad\qquad\qquad \mathit{ite}\ (\mathit{leq?}\ (\mathit{headIStr}[A](\pi_1\ ss))\ (\mathit{headIStr}[A](\pi_2\ ss)))\\
&\qquad\qquad\qquad\quad \langle\langle \mathit{headIStr}[A](\pi_1\ ss), \langle \mathit{tailIStr}[A](\pi_1\ ss), \pi_2\ ss\rangle\rangle,\\
&\qquad\qquad\qquad\quad\ \langle \mathit{headIStr}[A](\pi_2\ ss), \langle \pi_1\ ss, \mathit{tailIStr}[A](\pi_2\ ss)\rangle\rangle\rangle)
\end{aligned}$$

The type $\mathit{Unit} \equiv \nu(\lambda X.X)$ is the dual to Void and thus isomorphic to $\mathbf{1}$. Adapting examples from the isomorphic type $\mathit{InfStream}_{\mathbf{1}}$,

$$\begin{aligned}
&\mathit{unit} \equiv \mathbf{G}[\mathbf{1}]\mathit{Id}^{\mathbf{1}}\ *\\
&\mathit{Id}^{\mathit{Unit}} \cong \mathbf{out}^{-1} \cong \mathbf{G}[\mathit{Unit}]\mathbf{out} \cong \mathbf{out}
\end{aligned}$$

Many more inductive type examples can be adapted from [3].

B Translation to F_2

Since F_2 has figured prominently in the work on (co-)inductive types, we give a translation $\underline{\cdot}$ from $\lambda^{MM\mu\nu}$ to F_2. Type and term variables occurring only on the right-hand side of the equations are assumed to be fresh.

$$\begin{array}{rcl}
\underline{X} & \equiv & X\\
\underline{\mathbf{1}} & \equiv & \forall X.X \to X\\
\underline{\sigma_1 \times \sigma_2} & \equiv & \forall X.(\underline{\sigma_1} \to \underline{\sigma_2} \to) \to X\\
\underline{\sigma_1 + \sigma_2} & \equiv & \forall X.(\underline{\sigma_1} \to X) \to (\underline{\sigma_2} \to X) \to X\\
\underline{\sigma \to \sigma'} & \equiv & \underline{\sigma} \to \underline{\sigma'}\\
\underline{\mu(\lambda X.\tau)} & \equiv & \forall X.(\underline{\tau} \to X) \to X\\
\underline{\nu(\lambda X.\tau)} & \equiv & \forall Y.(\forall X.(X \to \underline{\tau}) \to X \to Y) \to Y\\
\underline{\forall X.\sigma} & \equiv & \forall X.\underline{\sigma}
\end{array}$$

$$\begin{array}{rcll}
\underline{x} & \equiv & x & \\
\underline{*} & \equiv & \Lambda X.\lambda x : X.x & \\
\underline{\langle t_1, t_2\rangle} & \equiv & \Lambda X.\lambda p : \underline{\sigma_1} \to \underline{\sigma_2} \to X.p\ \underline{t_1}\ \underline{t_2} & (\text{if } t_i : \sigma_i)\\
\underline{\pi_1 t} & \equiv & \underline{t}[\underline{\sigma_1}](\lambda l : \underline{\sigma_1}.\lambda r : \underline{\sigma_2}.l) & (\text{if } t : \sigma_1 \times \sigma_2)\\
\underline{\pi_2 t} & \equiv & \underline{t}[\underline{\sigma_2}](\lambda l : \underline{\sigma_1}.\lambda r : \underline{\sigma_2}.r) & (\text{if } t : \sigma_1 \times \sigma_2)\\
\underline{\mathbf{inl}^{\sigma_2} t} & \equiv & \Lambda X.\lambda l : \underline{\sigma_1} \to X.\lambda r : \underline{\sigma_2} \to X.l\ \underline{t} & (\text{if } t : \sigma_1)\\
\underline{\mathbf{inr}^{\sigma_1} t} & \equiv & \Lambda X.\lambda l : \underline{\sigma_1} \to X.\lambda r : \underline{\sigma_2} \to X.r\ \underline{t} & (\text{if } t : \sigma_2)\\
\underline{\mathbf{case}(t, t_1, t_2)} & \equiv & \underline{t}[\underline{\sigma}]\underline{t_1}\ \underline{t_2} & (\text{if } t_i : \sigma_i \to \sigma)\\
\underline{\lambda x : \sigma.t} & \equiv & \lambda x : \underline{\sigma}.\underline{t} & \\
\underline{\Lambda X.t} & \equiv & \Lambda X.\underline{t} &
\end{array}$$

$$
\begin{aligned}
\underline{i_1\ t_2} &\equiv \underline{t_1}\ \underline{t_2} \\
\underline{t[\tau]} &\equiv \underline{t}[\underline{\tau}] \\
\underline{\mathbf{in}\{u\}} &\equiv \lambda h : \underline{u\ \mu(u)}.\Lambda Y.\lambda f : \underline{u\ Y} \to Y. \\
&\qquad f\underline{(\Phi\{u\}[\mu(u)][Y]\ (\mathbf{R}\{u\}[Y]f)\ h)} \\
\underline{\mathbf{in}^{-1}\{u\}} &\equiv \lambda t : \underline{u\ (u\ \mu(u))}.\underline{\mathbf{R}\{u\}[u\ \mu(u)](\Phi\{u\}[u\ \mu(u)][\mu(u)]\ \mathbf{in}\{u\}\ t)} \\
\underline{\mathbf{R}\{u\}[\tau]f} &\equiv \lambda t : \underline{\mu(u)}.t[\underline{\tau}]\underline{f} \\
\underline{\mathbf{out}\{u\}} &\equiv \lambda t : \underline{\nu(u)}.t[\underline{u\ \nu(u)}](\Lambda Y.\lambda f : Y \to \underline{u\ Y}.\lambda x : Y. \\
&\qquad\qquad \underline{\Phi\{u\}[Y][\nu(u)]\ (\mathbf{G}\{u\}[Y]f)\ (f\ x)}) \\
\underline{\mathbf{out}^{-1}\{u\}} &\equiv \lambda t : \underline{u\ \nu(u)}.\underline{\mathbf{G}\{u\}[u\ \nu(u)](\Phi\{u\}[\nu(u)][u\ \nu(u)]\ \mathbf{out}\{u\}\ t)} \\
\underline{\mathbf{G}\{u\}[\tau]f} &\equiv \lambda x : \underline{\tau}.\Lambda Z.\lambda h : (\forall W.(W \to \underline{u\ W}) \to W \to Z.h[\underline{\tau}]\underline{f}\ x
\end{aligned}
$$

This translation maps $\lambda^{MM\mu\nu}$ types into F_2 types which are closely related to the standard F_2 encodings of these types. For example,

$$\underline{Nat} \equiv \forall Z.((\forall X.X \to X) \to Z) \to (Z \to Z) \to Z$$

which is isomorphic to $\forall Z.Z \to (Z \to Z) \to Z$, the normal definition.

Naturally, using the definition of *pred* in Example 6, $\underline{pred}$ is similar to the standard F_2 definition. However, translating Example 11 (and simplifying with isomorphisms for readability) results in an alternative:

$$
\begin{aligned}
\underline{pred} \cong_{F_2} \lambda n : \underline{Nat}.(n[&\forall Z.Z \to (\underline{Nat} \to Z) \to Z] \\
&(\Lambda Z.\lambda z : Z.\lambda s : \underline{Nat} \to Z.z) \\
&(\lambda y : (\forall Z.Z \to (\underline{Nat} \to Z) \to Z). \\
&\quad (\Lambda Z.\lambda z : Z.\lambda s : \underline{Nat} \to Z. \\
&\qquad s(y[\forall Z.Z \to (\underline{Nat} \to Z) \to Z]\ \underline{0}\ \underline{succ})))) \\
&[\underline{Nat}]\ \underline{0}\ \underline{Id^{Nat}}
\end{aligned}
$$

However, even using this definition, it requires linear-time to evaluate $\underline{pred}\ n$, since F_2 does not have an one-step equivalent of $\mathbf{in}^{-1}(\mathbf{in}\ t) \longrightarrow t$.

References

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press. 1985.

[2] Peter Aczel and Max Mendler. *A Final Coalgebra Theorem*. In *Category Theory and Computer Science*, Lecture Notes in Computer Science 389, eds. D. H. Pitt, D. E. Rydeheard, P. Dybjer, A. M. Pitts, and A. Poigné, pages 357–365. September 1989.

[3] Richard S. Bird. *Lectures on constructive functional programming.* Technical monograph PRG-69, Oxford University Computing Laboratory. 1988.

[4] R. M. Burstall. *Inductively Defined Functions in Functional Programming Languages.* Technical report LFCS-87-25, University of Edinburgh. 1987.

[5] Thierry Coquand and Christine Paulin. *Inductively defined types.* In *COLOG-88*, Lecture Notes in Computer Science 417, eds. P. Martin-Löf and G. Mints, pages 50–66. December 1988.

[6] Roy L. Crole. *Recursive Types via Fixpoint Objects.* Unpublished manuscript. 1992.

[7] M. M. Fokkinga, E. Meijer. *Program calculation properties of continuous algebras.* Technical report CS-R9104, Computer Science/Department of Algorithmics and Architecure, CWI. January 1991.

[8] Tatsuya Hagino. *A Typed Lambda Calculus with Categorical Type Constructors.* In *Category Theory and Computer Science*, Lecture Notes in Computer Science 283, eds. D. H. Pitt, A. Poingé, and D. E. Rydeheard, pages 140–157. September 1987.

[9] Tatsuya Hagino. *A Categorical Programming Language.* Ph.D. thesis. University of Edinburgh. September 1987.

[10] Tatsuya Hagino. *Codatatypes in ML.* In *Journal of Symbolic Computation*, 8, pages 629–650. 1989.

[11] Robert Harper, John C. Mitchell, and Eugenio Moggi. *Higher-Order Modules and the Phase Distinction.* In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 341–354. January 1990.

[12] Daniel Leivant. *Contracting Proofs to Programs.* In *Logic and Computer Science*, Lecture Notes in Mathematics 1429, ed. P. Odifreddi, pages 279–327. 1990.

[13] Daniel Leivant. *Finitely stratified polymorphism.* Technical report CMU-CS-90-160, Carnegie Mellon University. August 1990.

[14] Mark Lillibridge. Personal communication. July 1991.

[15] Erik Meijer, Maarten Fokkinga, Ross Paterson. *Functional Programming with Banana, Lenses, Envelopes and Barbed Wire.* In *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 523, ed. John Hughes, pages 124–144. 1991.

[16] N. P. Mendler. *First- and Second-Order Lambda Calculi with Recursive Types.* Technical report 86-174, Cornell University. July 1986.

[17] N. P. Mendler. *Recursive Types and Type Constraints in Second-Order Lambda Calculus.* In *Proceedings of the 2nd Symposium on Logic in Computer Science*, pages 30–36. June 1987.

[18] Paul Francis Mendler. *Inductive Definition in Type Theory.* Technical report 87-870, Cornell University. September 1987.

[19] Nax Paul Mendler. *Quotient types via coequalizers in Martin-Lof type theory.* In *Proceedings of the Logical Frameworks Workshop*, pages 349–361. May 1990.

[20] Nax Paul Mendler. *Predicative Type Universes and Primitive Recursion.* In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 173–184. June 1991.

[21] Spiro Michaylov and Frank Pfenning. *Compiling the Polymorphic λ-Calculus.* In *Proceedings of the ACM/IFIP Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 285–296. June 1991.

[22] Michel Parigot. *Programming With Proofs: A Second Order Type Theory.* In *Proceedings of the 2nd European Symposium on Programming*, ed. H. Ganzinger, pages 145–159. 1988.

[23] Michel Parigot. *Recursive Programming With Proofs.* Unpublished manuscript. 1988.

[24] Michel Parigot. *On the Representation of Data in Lambda-Calculus.* In *Proceedings of the 3rd Workshop on Computer Science Logic*, Lecture Notes in Computer Science 440, eds. E. Börger, H. Kleine Büning, M. M. Richter, pages 309–321. October 1989.

[25] Frank Pfenning and Christine Paulin-Mohring. *Inductively Defined Types in the Calculus of Constructions.* In *Proceedings of the Fifth Conference on the Mathematical Foundations of Programming Semantics*, Lecture Notes in Computer Science 442, eds. M. Main, A. Melton, M. Mislove, and D. Schmidt, pages 209–228. March 1989.

[26] Benjamin Pierce, Scott Dietzen, and Spiro Michaylov. *Programming in Higher-Order Typed Lambda-Calculi.* Technical report CMU-CS-89-111, Carnegie Mellon University. March 1989.

[27] G. C. Wraith. *A Note of Categorical Datatypes.* In *Category Theory and Computer Science*, Lecture Notes in Computer Science 389, eds. D. H. Pitt, D. E. Rydeheard, P. Dybjer, A. M. Pitts, and A. Poigné, pages 118–127. September 1989.